EECS 4101/5101

Prof. Andy Mirzaian

Priority Queues

TOPICS

- Priority Queues
- Leftist Heaps
- Skew Heaps
- Binomial Heaps
- Fibonacci Heaps
- Recent Developments

References:

- [CLRS 2nd edition] chapters 19, 20 or
  [CLRS 3rd edition] chapter 19 & Problem 19-2 (pp:527-529)
- Lecture Notes 4, 5
- AAW animations

Priority Queues

Basic Priority Queues

- Each item x has an associated priority denoted key[x].
- Item priorities are not necessarily distinct and may be time varying.
- A Priority Queue Q is a set of prioritized data items that supports the following basic priority queue operations:

  Insert(x,Q): insert (new) item x into Q. (Duplicate priorities allowed.)

  DeleteMin(Q): remove and return the minimum key item from Q.

- Notes:
  1. Priority Queues do not support the Dictionary Search operation.
  2. DeleteMin for min-PQ: the lower the key the higher the priority.
  3. DeleteMax for max-PQ: the higher the key the higher the priority.
  4. More PQ operations shortly.
- Example:

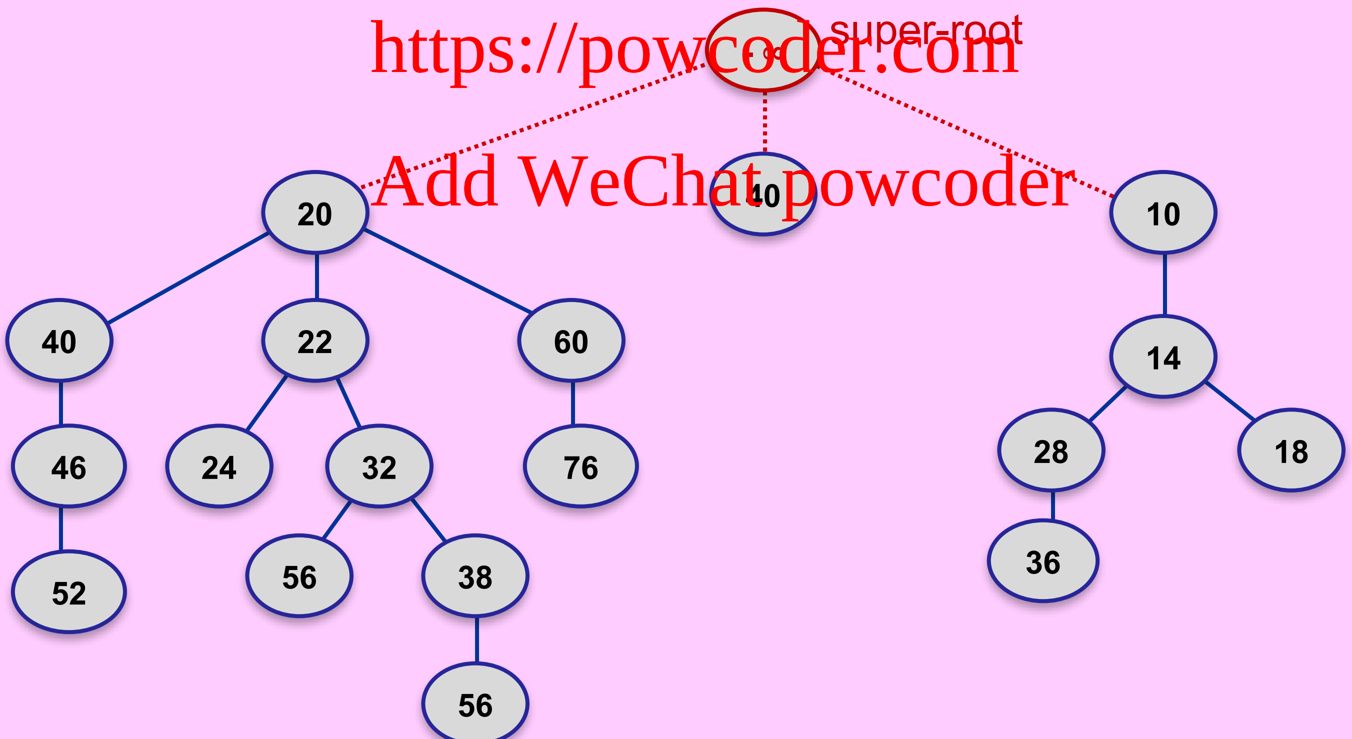
  An ordinary queue can be viewed as a priority queue where the priority of an item is its insertion time.

HEAP

- **Heap Ordered Tree:**
  Let T be a tree that holds one item per node.
  T is (min-) heap-ordered iff $\forall$nodes x $\neq$ root(T): key[x] key[parent(x)].

- [Note: Any subtree of a heap-ordered tree is heap-ordered.]

- **Heap:** a forest of one or more node-disjoint heap-ordered trees.



Some Applications

- Sorting and Selection.
- Scheduling processes with priorities.
- Priority driven discrete event simulation.
- Many Graph and Network Flow algorithms, e.g.,
  Prim's Minimum Spanning Tree algorithm,
  Dijkstra's Shortest Paths algorithm,
  Max Flow,
  Min-Cost Flow,
  Weighted Matching, …
- Many problems in Computational Geometry, e.g.,
  plane sweep (when not all events are known in advance).

More Heap Operations

Mergeable Heap Operations:

MakeHeap(e): generate and return a heap that contains the single item e with a given key key[e].

Insert(e,H): insert (new) item e, with its key key[e], into heap H.

FindMin(H): return the minimum key item from heap H.

DeleteMin(H): remove and return the minimum key item from heap H.

Union(H_1, H_2): return the heap that results from replacing heaps H_1 and H_2 by their disjoint union $H_1 \cup H_2$.
(This destroys H_1 and H_2.)

DecreaseKey(x,K,H): Given access to node x of heap H with key[x] > K, decrease Key[x] to K and update heap H.

Delete(x,H): Given access to node x of heap H, remove the item at node x from Heap H.

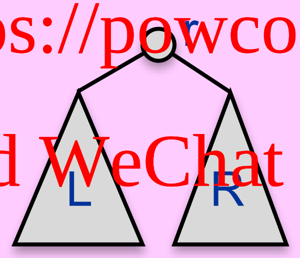
Beyond Standard Heap

Williams[1964]: Array based binary heap (for HeapSort). See [CLRS ch 6].
Time complexities of mergeable heap operations on this structure are:

O(1)	MakeHeap(e), FindMin(H).	
O(log n)	Insert(e,H), DeleteMin(H)	(n = |H|)
O(n)	Union(H_1, H_2)	(n = $\|H_1\| + \|H_2\|$)

Insert and DeleteMin via Union on a pointer-based (binary tree) heap:

Insert(e,H)
  H’ ← MakeHeap(e)
  H ← Union(H, H’)
end



DeleteMin(H)
  r ← root[H]
  if r = nil **then return** error
  MinKey ← Key[r]
  root[H] ← Union(left[r], right[r])
  return MinKey
end

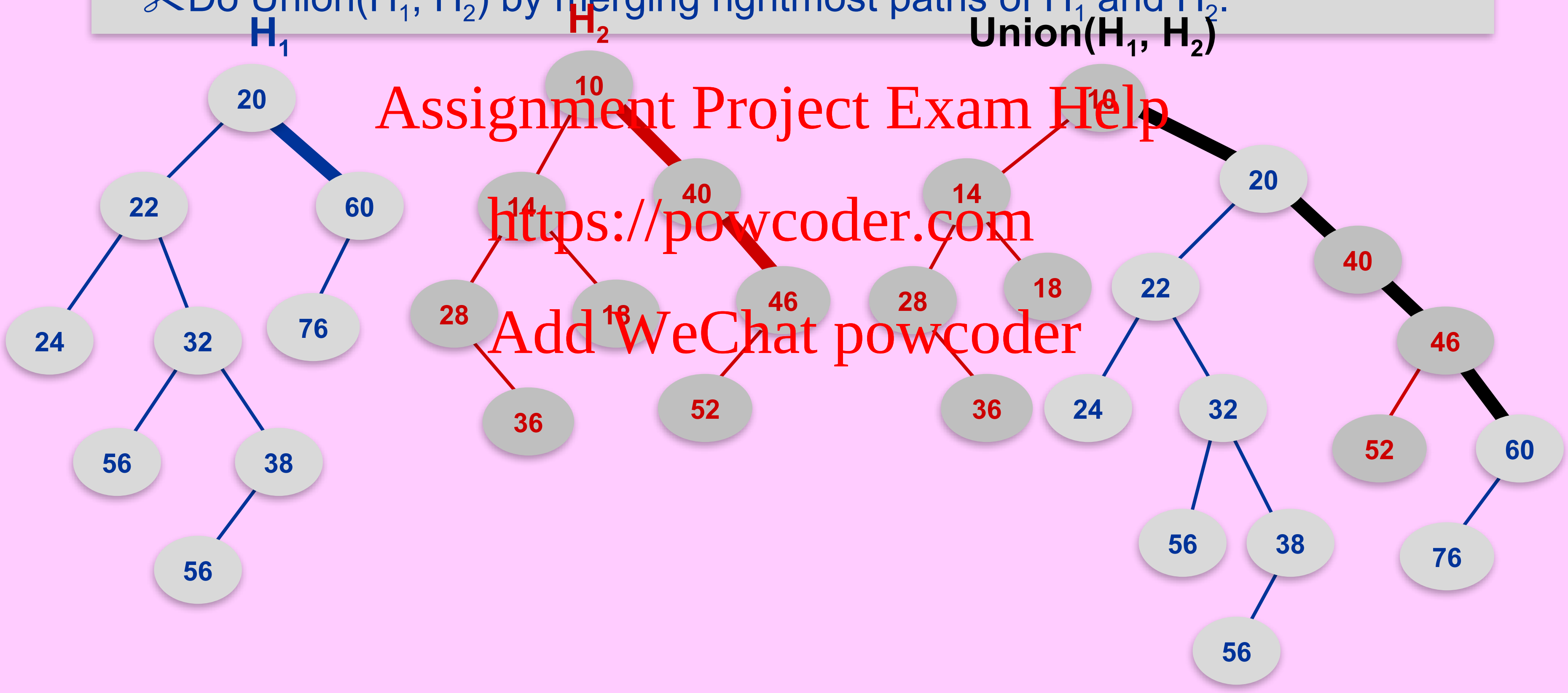
- ✂ Can we improve Union in order to improve both Insert & DeleteMin?
- ✂ Can we do both Insert & DeleteMin in o(log n) time, even amortized?
  No. That would violate the Ω(n log n) sorting lower-bound. Why?
- ✂ How about alternative trade offs? Amortization? Self-adjustment? …

Leftist Heap

Union on Pointer based Binary Heap

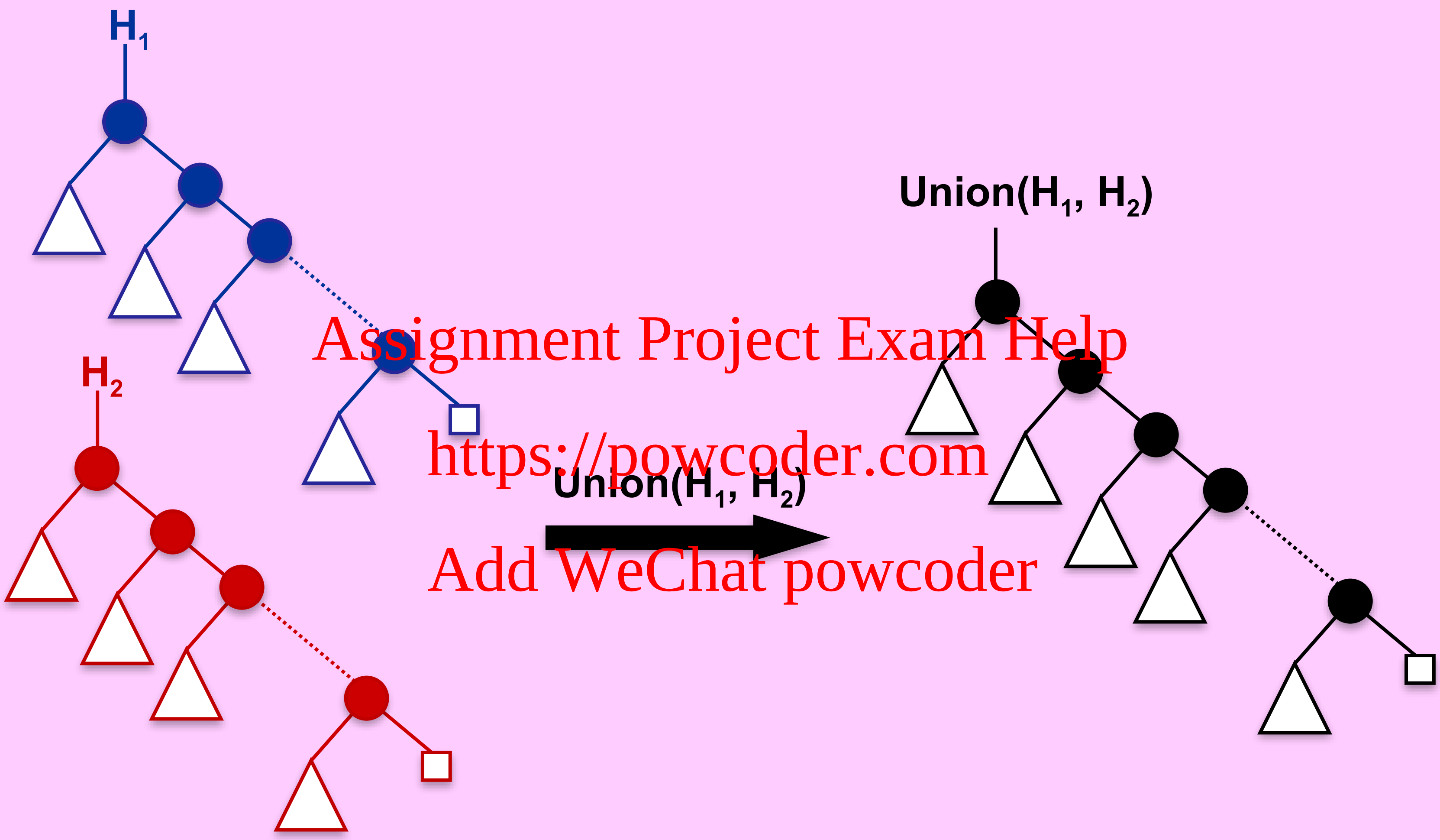
FIRST IDEA:

- In a heap-ordered tree, items on any path appear in sorted order.
- Two sorted lists can be merged in time linear in total size of the two lists.
- Do Union(H_1, H_2) by merging rightmost paths of H_1 and H_2.



Running Time = O(# nodes on the merged rightmost path).

Union by Merging Rightmost Paths



SECOND IDEA: Keep rightmost path of the tree a shortest path from root to any external node (i.e., minimize depth of rightmost external node). Recursively do so in every subtree.

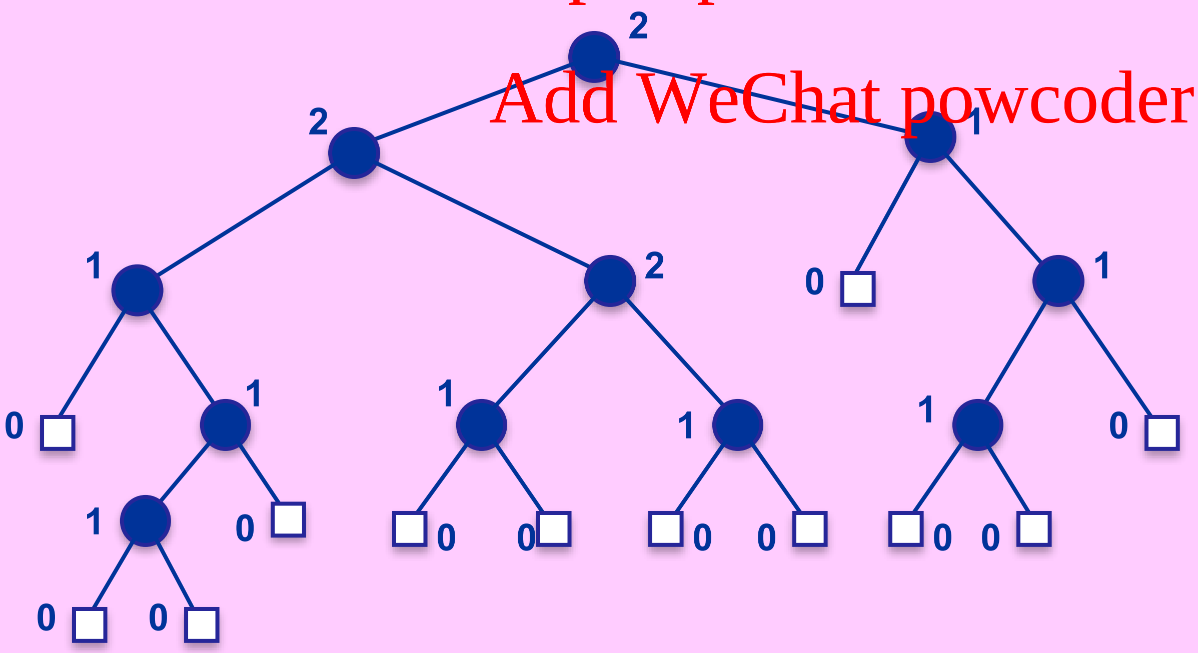
“dist” field

DEFINITION: For every node x:

min distance from x to a descendant external node.

Recurrence:

LN4 defines 1 here.

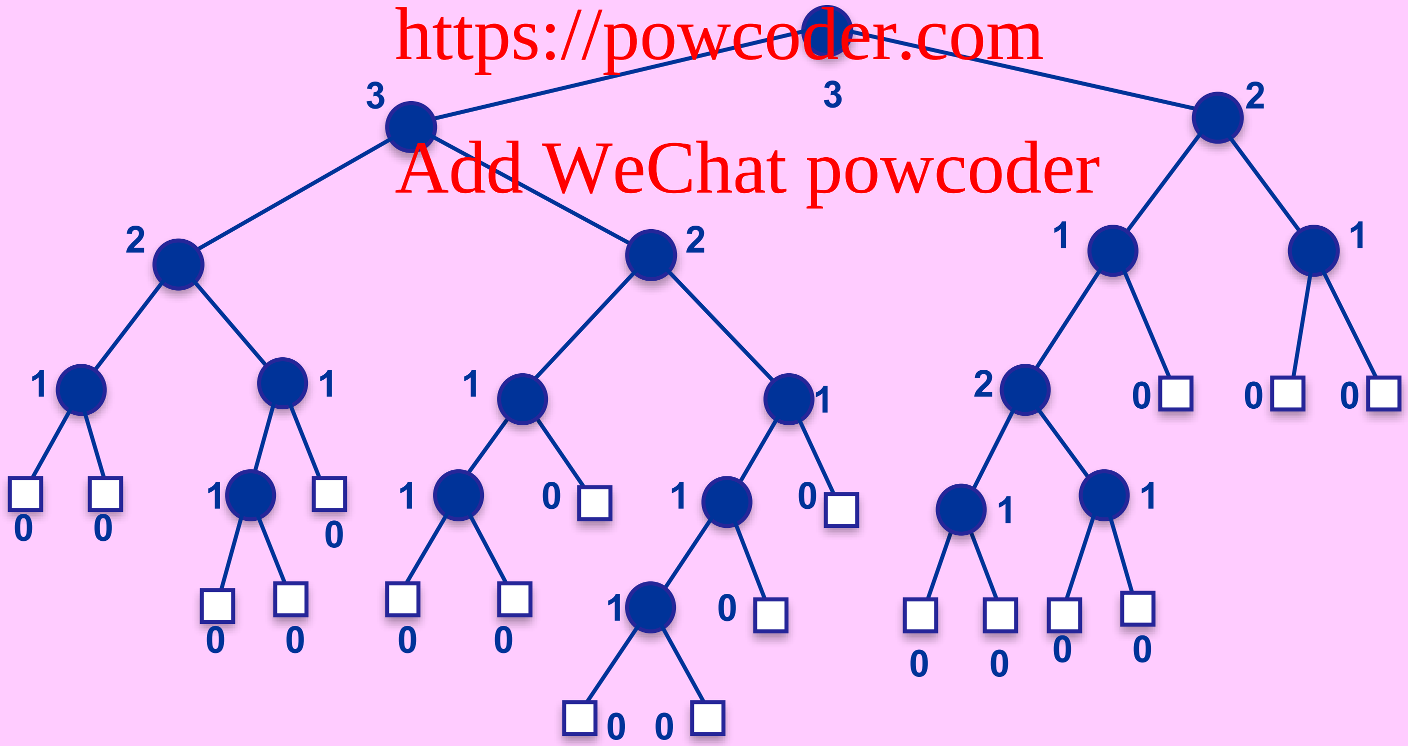


Leftist Tree

DEFINITION: A binary tree T is a **Leftist tree** if for every node x ≠nil in T:

Leftist Recurrence:

FACT: every subtree of a Leftist tree is a Leftist tree.



Leftist Tree has short rightmost path

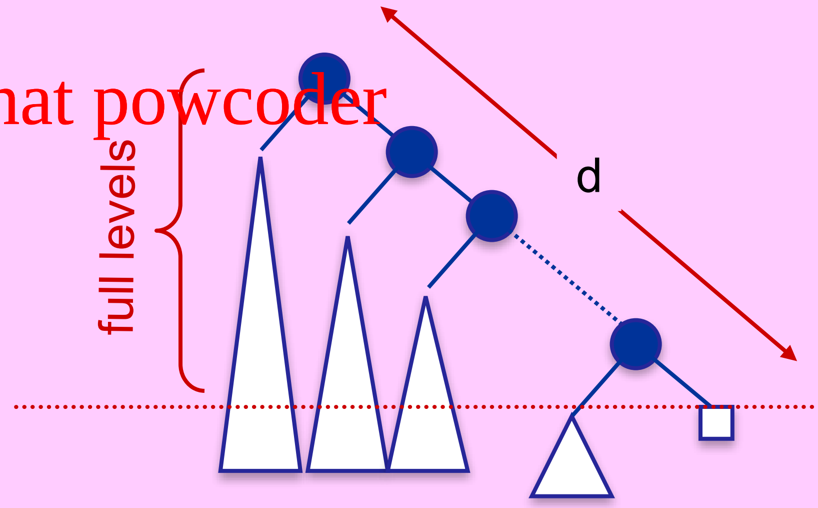
FACT: An n-node leftist tree has $\leq \log(n+1)$ nodes on its rightmost path.

Proof:

If d = depth of the rightmost external node,
then every external node has depth at least d.

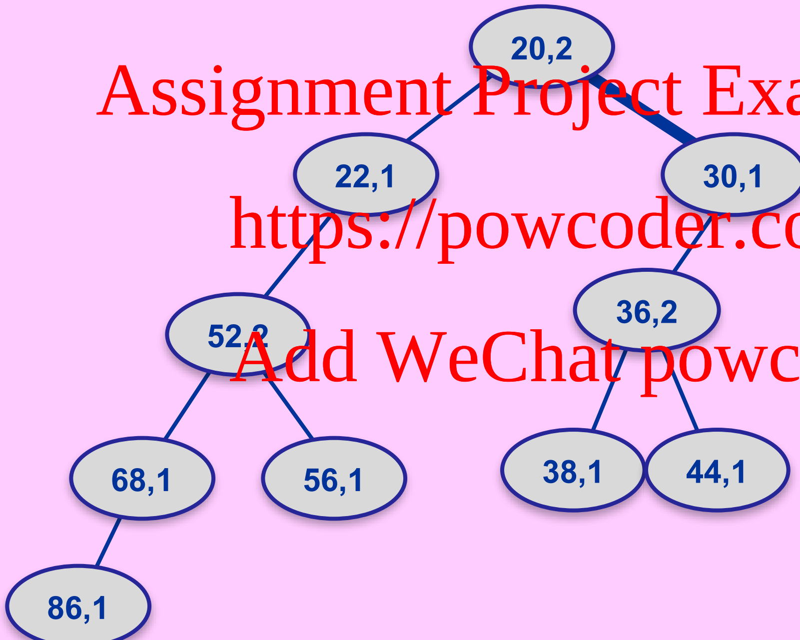
$$n \quad 2^0 + 2^1 + 2^2 + \cdots + 2^{d-1} = 2^d - 1$$

$$\Rightarrow \quad d \leq \log(n+1).$$



Leftist Heap

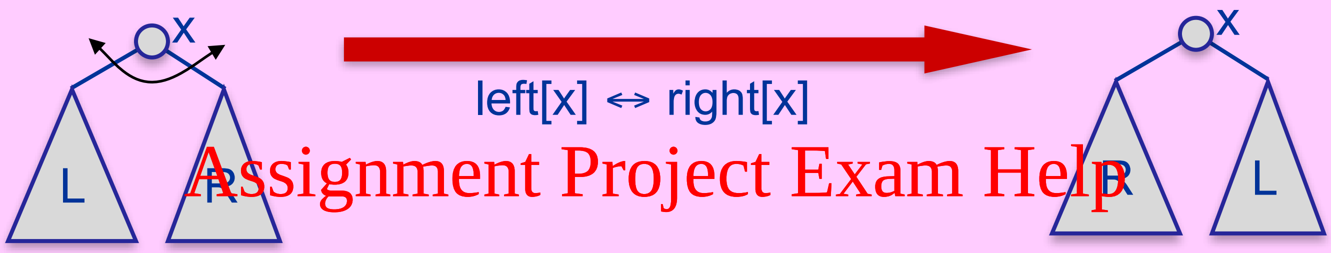
DEFINITION: A **Leftist heap** is a heap-ordered leftist binary tree, where each node x in it explicitly stores dist[x], as well as holding an item with priority key[x].

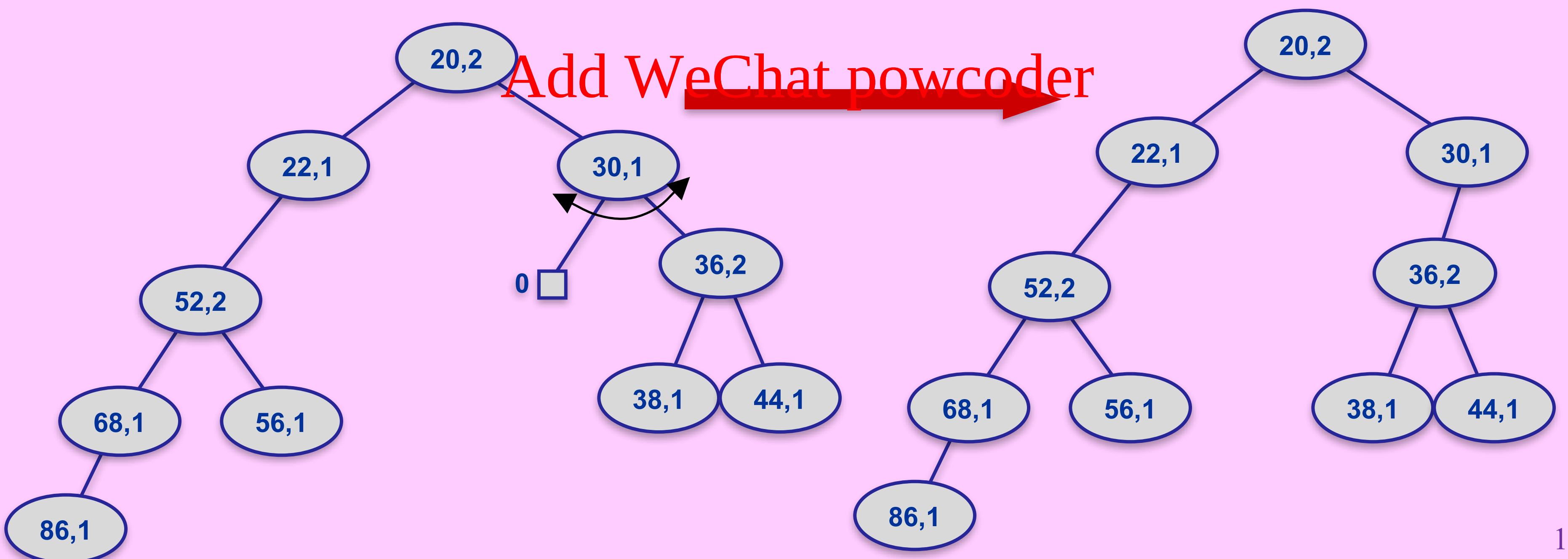


In each node, 1st # is key, 2nd # is dist.

Child swap

To restore leftist property at node x, swap its left & right child pointers.
This O(1) time operation preserves the heap-order property.





UNION

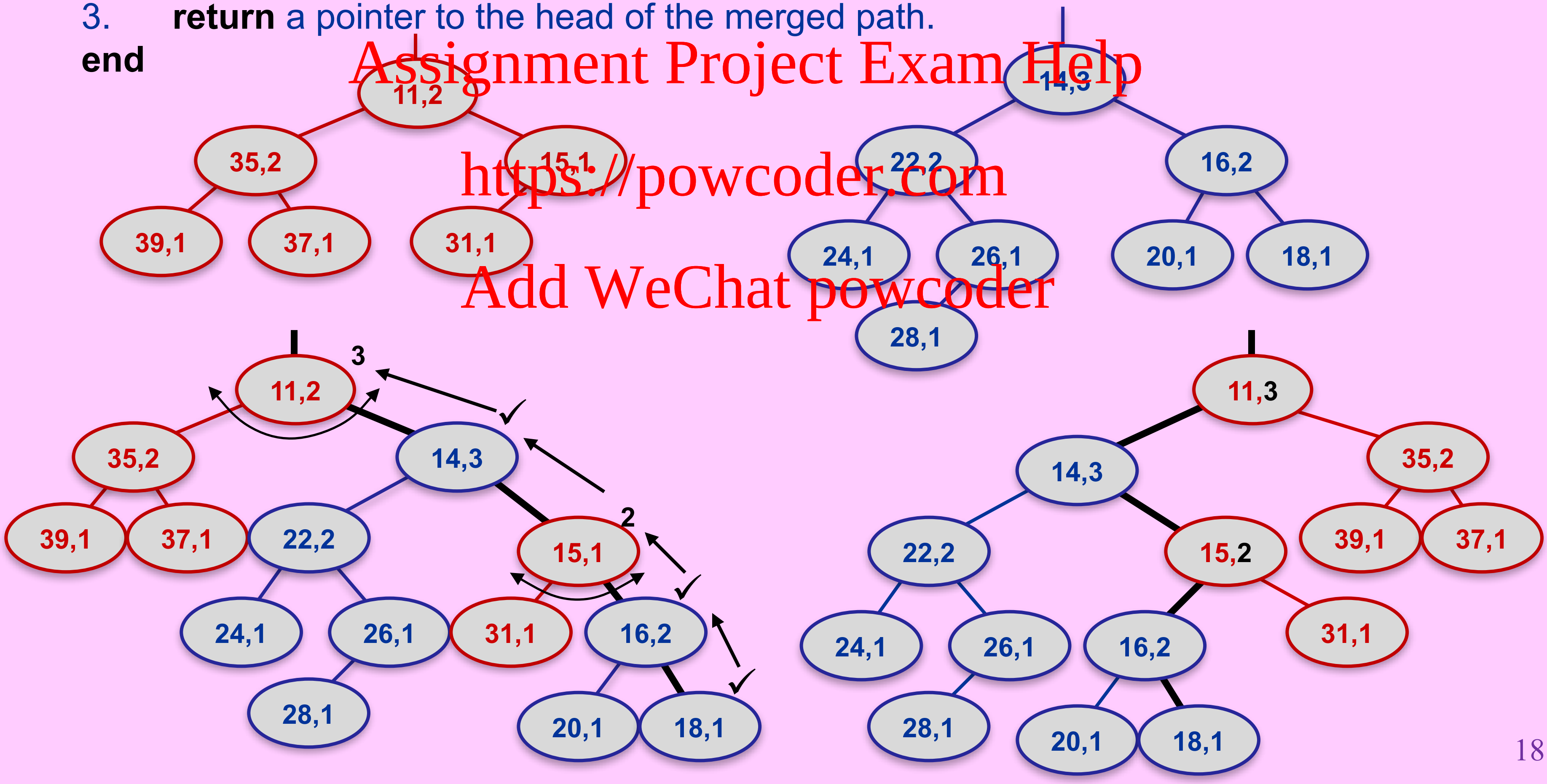
Union (H_1 , H_2)

1. First pass down: merge rightmost paths of H_1 and H_2.
2. Second pass up: for each node x on the merged path upwards do:

    2a. if needed, swap left/right child pointers of x to restore the leftist property.

    2b. update dist[x].
3. **return** a pointer to the head of the merged path.

end

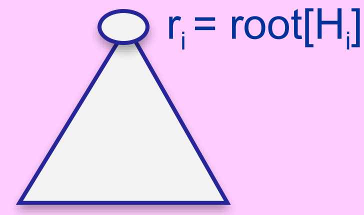


Recursive Union

Union(r_1 , r_2) (* r_1, r_2 are heap roots *)

1. **if** r_1 = nil **then return** r_2
2. **if** r_2 = nil **then return** r_1
3. **if** key[r_1] > key[r_2] **then** $r_1 \leftrightarrow r_2$
4. right[r_1] ← Union(right[r_1] , r_2)
5. **if** D(right[r_1]) > D(left[r_1]) **then** right[r_1] ↔ left[r_1]
6. dist[r_1] ← D(right[r_1]) + 1
7. **return** r_1

end



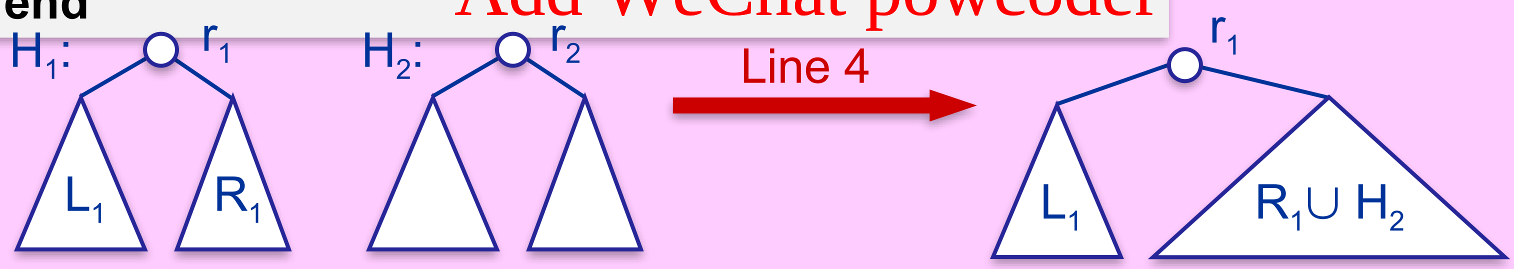
D(x)

if x = nil

then return 0

return dist[x]

end



Running time of UNION (H_1 , H_2):

$O(\log(n_1+1) + \log(n_2+1)) = O(\log n)$. [$n_1 = |H_1|$, $n_2 = |H_2|$, $n = n_1 + n_2$.]

By LOG Lemma: $\log(n_1+1) + \log(n_2+1) \le 2\log(n+2) - 2 \le 2\log n$, $\forall n>1$.

MAKEHEAP, INSERT, DELETEMIN

MakeHeap(e)

1. r ← new node()
2. key[r] ← key[e]   (* also store any secondary fields of e in r *)
3. dist[r] ← 1;   p[r] ← left[r] ← right[r] ← nil
4. **return** r

end

Insert(e, r)   (* r = root of a heap *)

1. r' ← MakeHeap(e)
2. r ← UNION(r, r')

end

DeleteMin(r)

1. **if** r = nil **then return** error
2. MinKey ← key[r]
3. r ← UNION(left[r], right[r])
4. **return** MinKey

end

MakeHeap
O(1) time.

Insert and DeleteMin
O(log n) time.

Leftist Heap Complexity

THEOREM: Worst-case times of mergeable leftist heap operations are:

O(1) for MakeHeap, FindMin

O(log n) for Union, Insert, DeleteMin,

where n is the total size of the heaps involved in the operation.

Exercise: Show that leftist heap operations DecreaseKey and Delete can also be done in O(log n) worst-case time.

Leftist Heap Complexity

Operation	**Worst-case**
MakeHeap	O(1)
FindMin	O(1)
Union	O(log n)
Insert	O(log n)
DeleteMin	O(log n)
DecreaseKey	O(log n)
Delete	O(log n)

Skew Heap (a simpler self-adjusting version of leftist heap, discussed next) achieves these running times in the amortized sense.

Skew Heap:
Self-Adjusting version of Leftist Heap

Skew Heaps

- A Skew Heap is an arbitrary heap-ordered binary tree.
- Unlike Leftist heaps, nodes in skew heaps do not store the "dist" field.
- Self-adjusting feature is in the Union:

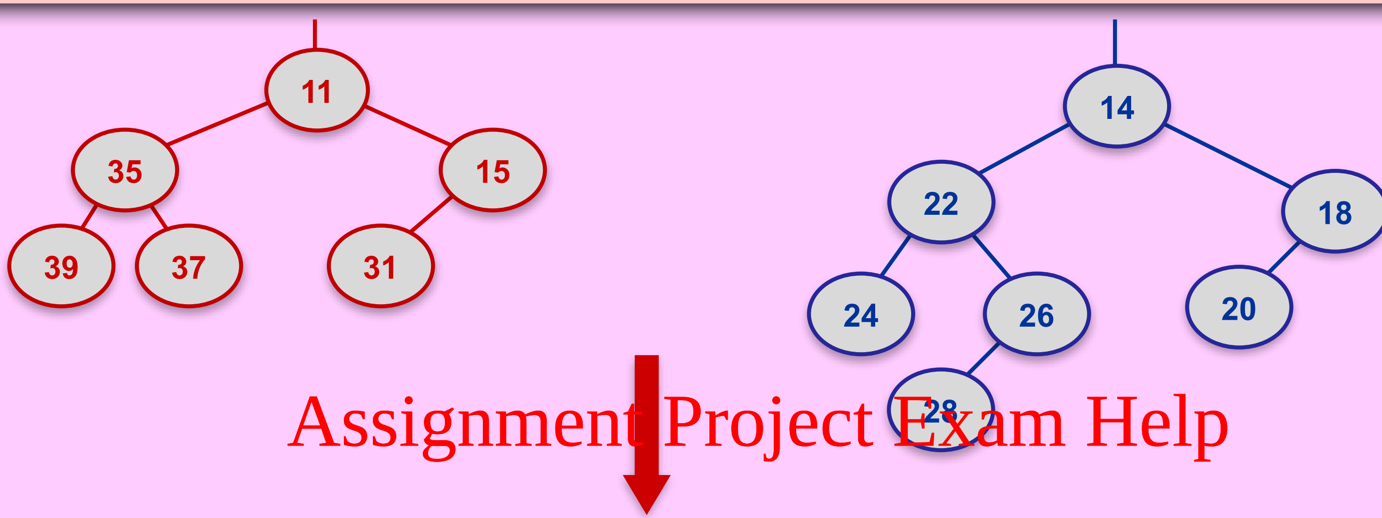
Union(H_1 , H_2)

1. Merge the rightmost paths of H_1 and H_2.
2. **for** each node x on the merged path except the lowest
   do swap left/right child pointers of x
3. **return** a pointer to the head of the merged path.

end

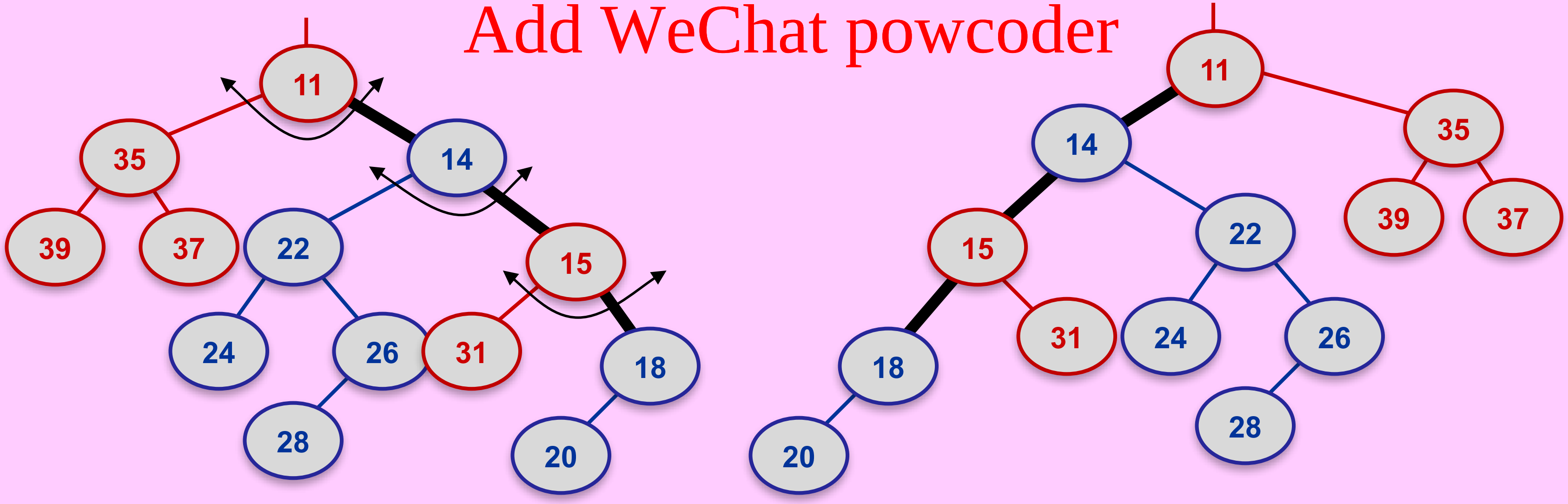
- This makes the longish merged path to become part of the leftmost path.
- In the amortized sense, this tends to keep rightmost paths short.
- Insert, DeleteMin are done via Union which dominates their amortized costs.

Skew Heap Union **Example**

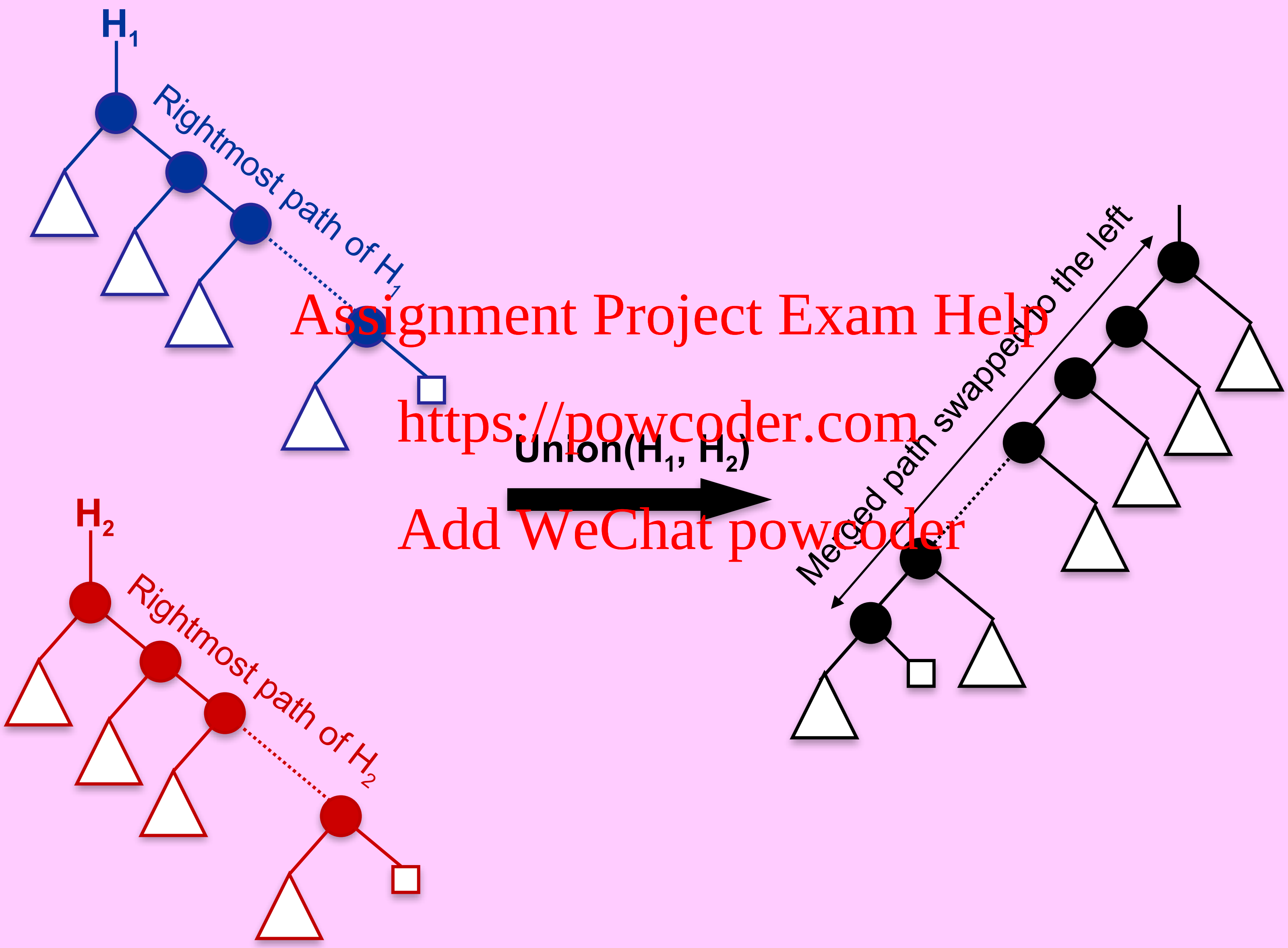


(a) Merge of rightmost paths

(b) Child-swap of nodes on the merged path



Skew Heap Union



Potential Function

- Weight of node x: w(x) = # descendants of x, inclusive.
- Non-root node x is HEAVY if w(x) > ½ w(parent(x)),

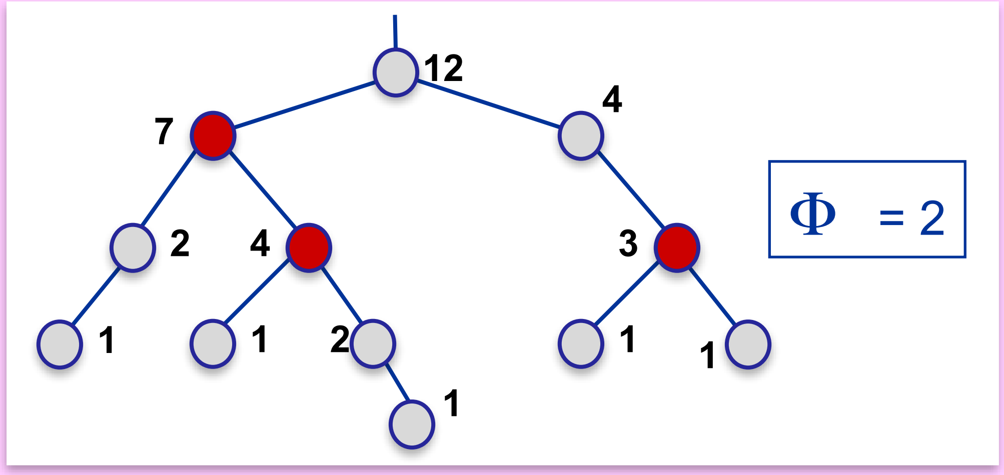
  LIGHT if w(x) ≤ ½ w(parent(x)),

  LEFT if x = left[parent(x)],

  RIGHT if x = right[parent(x)].

- Root is none of these.

- POTENTIAL: $\Phi(T)$ = number of RIGHT HEAVY nodes in T.



Heavy & Light Facts

HEAVY FACT: Among any group of siblings at most **one** can be heavy.

Proof: This applies to even non-binary trees:
Siblings x and y of parent p: $1 + w(x) + w(y) \le w(p)$.
$[w(x) > ½\, w(p)] \wedge [w(y) > ½\, w(p)] \Rightarrow w(x) + w(y) > w(p)$. A contradiction!

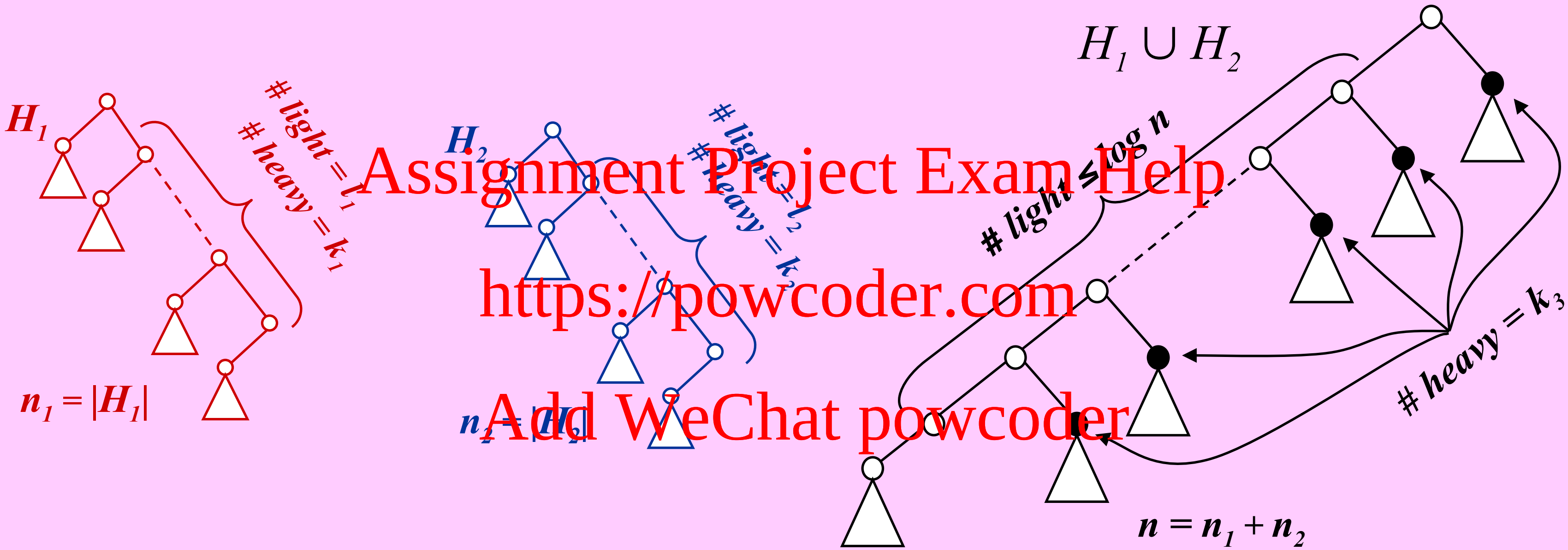
LIGHT FACT: In any n-node tree, any path has at most **log n** light nodes.

Proof: $\forall$non-root node x along the path: $1 \le w(x) < w(parent(x))$.
Descending along the path, node weights are monotonically reduced.
Furthermore, $w(x) \le ½\, w(parent(x))$ if x is light.
w(root)=n. If it's divided by half more than log n times, it would become < 1.

Amortized Analysis of Union

THEOREM: Amortized time of UNION (H_1,H_2) is $O(\log n)$, $n = |H_1|+|H_2|$.

Proof:



$\hat{c} = c + \Delta\Phi$

$= [\,(1+ l_1 + k_1) + (1+ l_2 + k_2)\,] + [k_3 - k_1 - k_2]$

$= 2 + l_1 + l_2 + k_3$

$\leq 2 + \log n_1 + \log n_2 + \log n$ [by Light & Heavy Facts]

$\leq 2 + 2 \log (n_1 + n_2) - 2 + \log n$ [by LOG Lemma]

Skew Heap Amortized Complexity

THEOREM: Amortized times of mergeable skew heap operations are:

O(1) for MakeHeap, FindMin

O(log n) for Union, Insert, DeleteMin,

where n is the total size of the heaps involved in the operation.

Exercise:

Show that skew heap operations DecreaseKey and Delete can also be done in O(log n) amortized time.

Skew Heap Complexity

Operation	Amortized
MakeHeap	O(1)
FindMin	O(1)
Union	O(log n)
Insert	O(log n)
DeleteMin	O(log n)
DecreaseKey	O(log n)
Delete	O(log n)

Binomial Heap (discussed next) improves these amortized times.

Binomial Heaps

Heap-ordered Multi-way Tree

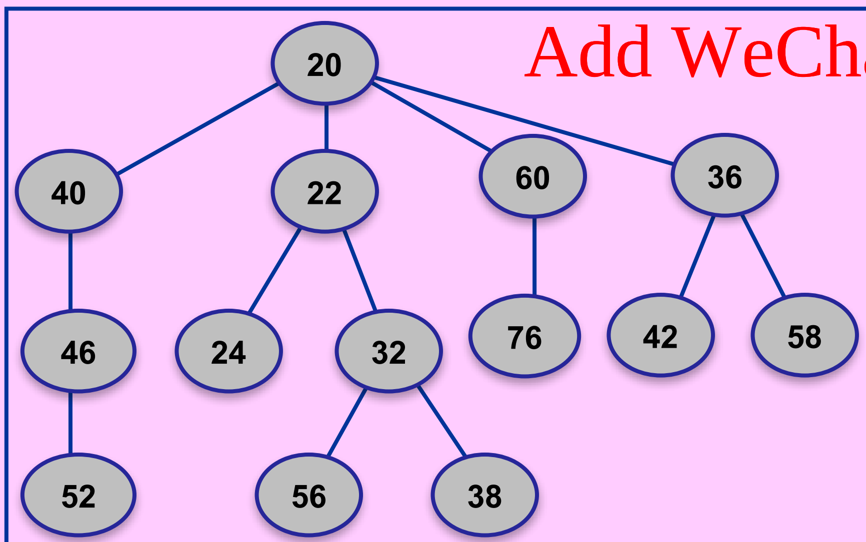
- Constant number of pointers per node suffice:
  Put children of any node x into a linked list using a right-sibling pointer at each child, and have x point to the head of that list, namely, its leftmost child.
  We may also use parent pointers.
  Any of these pointers may be nil (not shown in figure below.)
- For each node x we have:

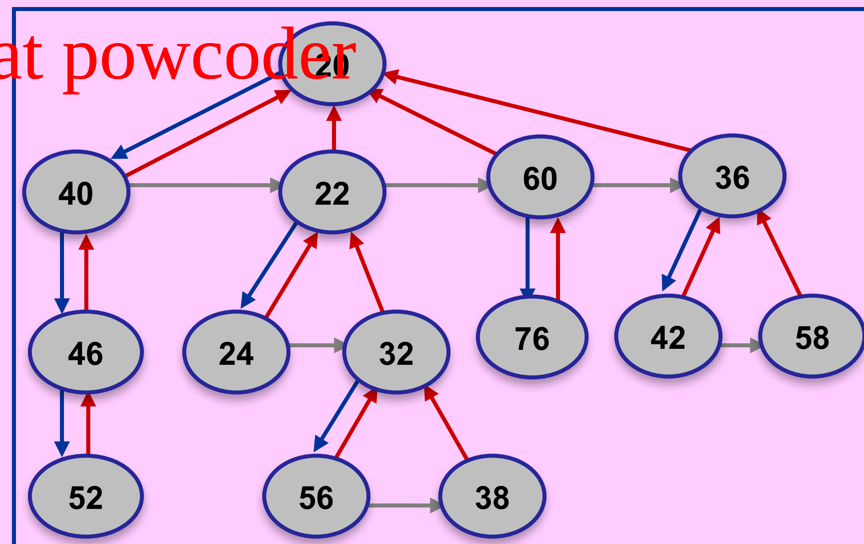
key[x] = (priority of) the item at node x

lchild[x] = (pointer to) leftmost child of x

rsib[x] = (pointer to) sibling of x immediately to its right

p[x] = (pointer to) parent of x

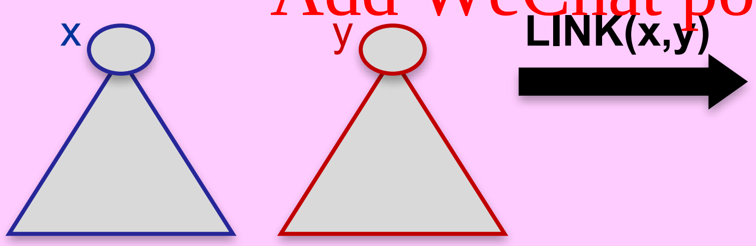


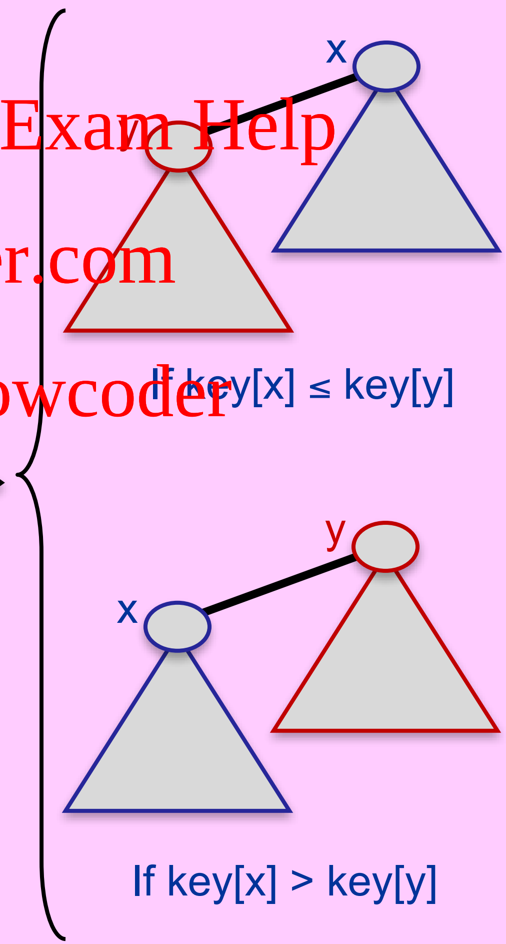


LINK operation

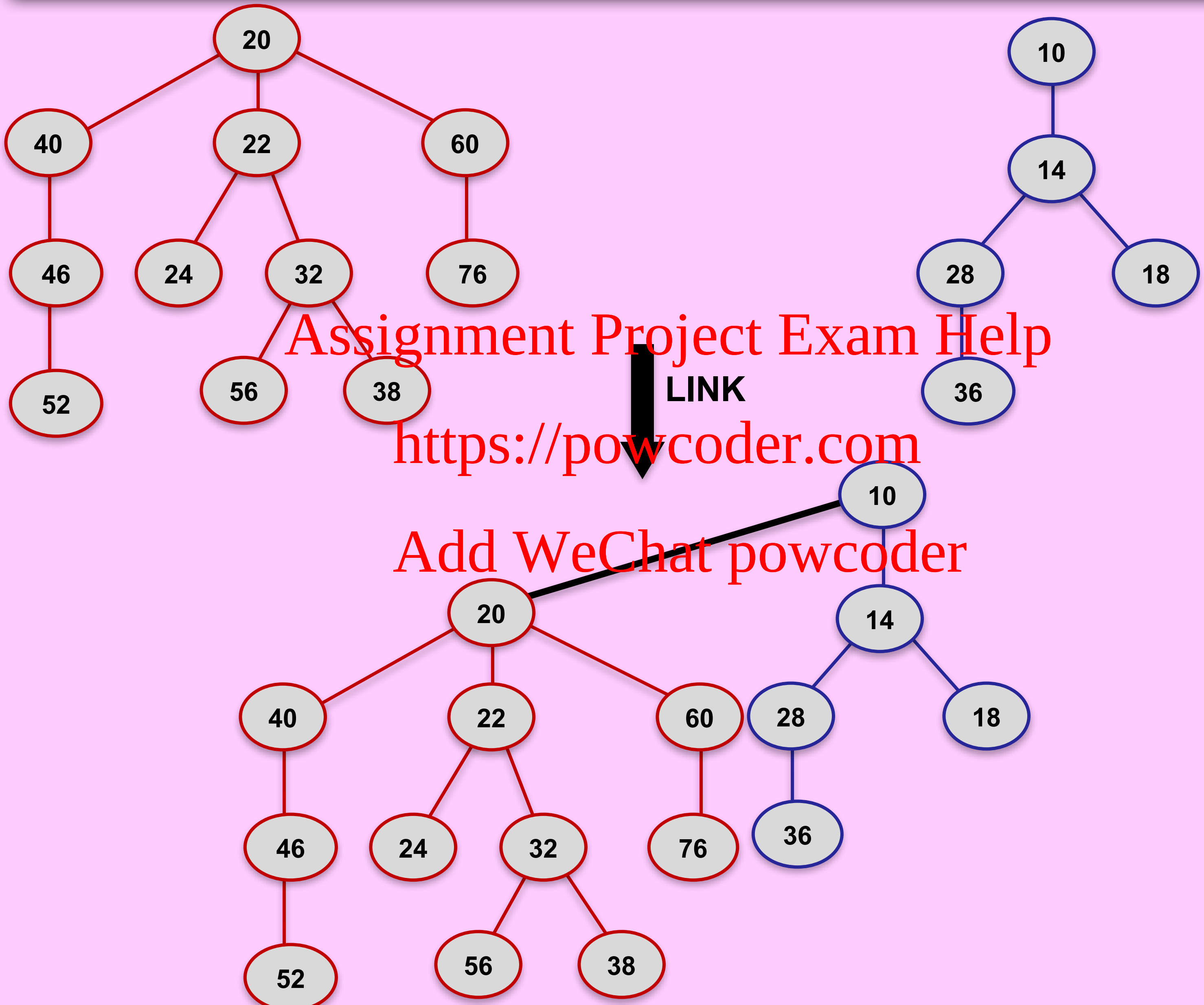
In O(1) time LINK joins two non-empty heap-ordered multi-way trees into one such tree.

LINK(x,y)
    if key[x] > key[y] **then** x ↔ y
    rsib[y] ← lchild[x]
    lchild[x] ← y
    p[y] ← x
    return x
end





Example


20
40
22
60
46
24
32
76
52
56
38
10
14
28
18
36
LINK
10
20
14
40
22
60
28
18
46
24
32
76
36
52
56
38

Other heap operations?

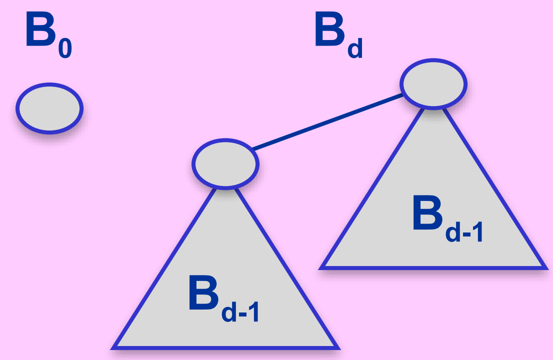
- Suppose "Our Heap" is a heap-ordered multi-way tree.
  - We can do UNION and INSERT in O(1) time using LINK.
  - Now a node may have up to O(n) children.

- How about DELETEMIN?
  - Detach min-key root from rest of the tree & return it at the end.
  - Also FINDMIN needs to know the new minimum key.
  - What to do with the (many) subtrees of the detached root?
  - We can use many LINKs to join them. That would be costly, O(n) time!

- TRADE OFF: instead of a single heap-ordered tree, let "Our Revised Heap" consist of a forest of (not too many) node-disjoint such trees, where their roots are linked together (using their already available right-sibling pointers). This is like having a "super root" with key = $-\infty$ as parent of all "regular roots".

- BINOMIAL HEAPS are a perfect fit for this setting.

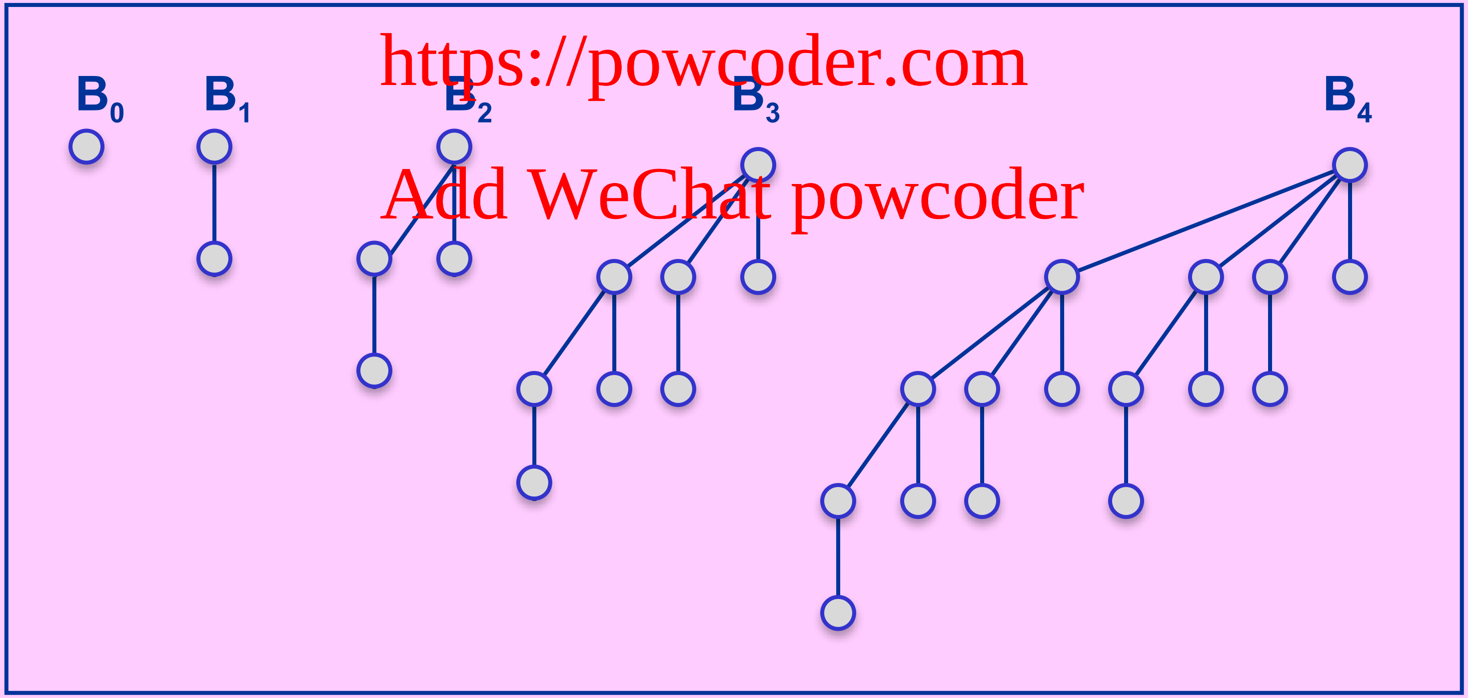
Binomial Trees



B_d = Binomial Tree of order (or degree) d:

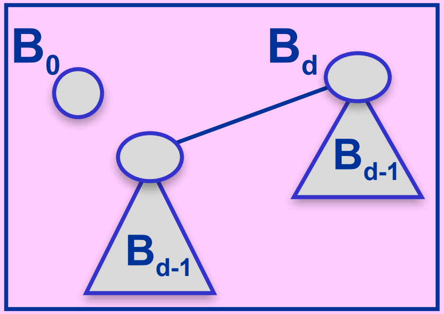
d = 0,1,2,3, …

Degree of a node = # of its children.

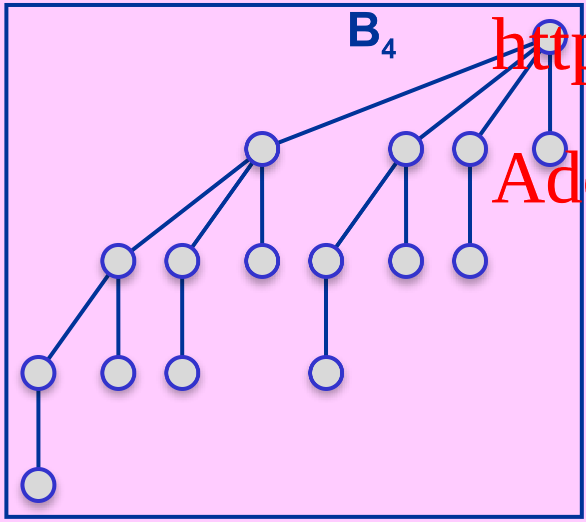


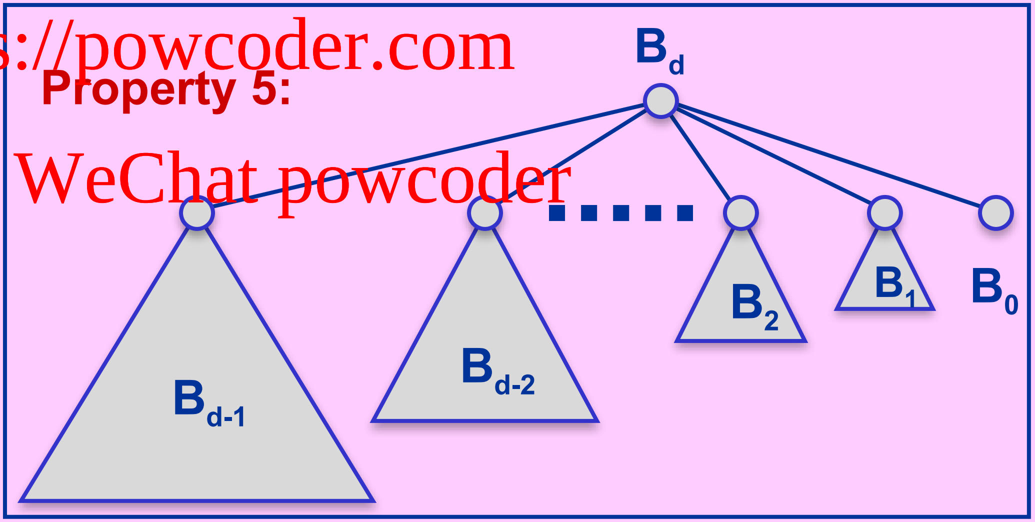
Binomial Tree Properties

FACT: $\mathbf{B_d}$ has the following properties:



1. Has 2^d nodes.
2. Has height d.
3. Has $\binom{d}{i}$ nodes at depth i, i = 0..d.
4. Root has degree d, every other node has degree $<$ d.
5. The d subtrees of the root, from right to left, are B_0, B_1, B_2, …, B_{d-1}.





COROLLARY (of Properties 1,2,4):

$\mathbf{B_d}$ has $\le$ n nodes $\Longrightarrow$ all its nodes have degree (& depth) $\le$ log n.

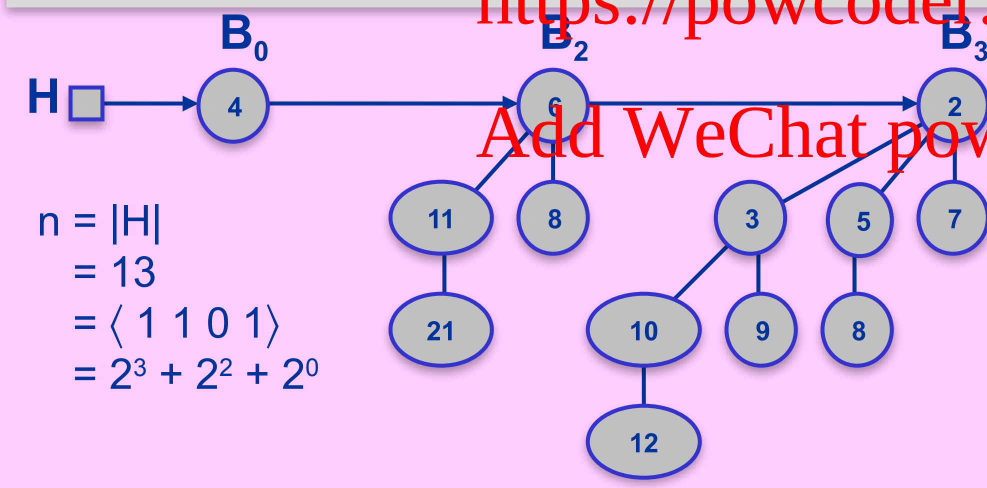
Binomial Heap

DEFINITION: Suppose binary representation of n is $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$, using $1+k = 1 + \lfloor \log n \rfloor$ 0/1 bits.

A **Binomial Heap H** of size n (# items) consists of a forest of node-disjoint heap-ordered Binomial trees of distinct root degrees, namely one B_d, for each $b_d = 1$, for $d = 0 \ldots \lfloor \log n \rfloor$.

The roots of these Binomial trees are sequentially linked (by right-sibling pointers) from smallest to largest tree (least-to-most significant bit).

Also, each node x of the Heap explicitly stores its degree deg[x].



$n = |H|$
$= 13$
$= \langle 1\ 1\ 0\ 1 \rangle$
$= 2^3 + 2^2 + 2^0$

Node x:

deg[x] = # children
key[x]
lchild[x]
rsib[x]
p[x]

FACT: An n-node Binomial Heap has at most 1+ log n trees, and all its nodes have degree, and depth, at most log n.

Binomial Heap vs Binary Counter

Even though Binomial Heaps do not explicitly do bit manipulation, their procedures resemble that of Binary Counter operations.

Binomial Heap	Binary Counter
LINK(x,y) deg[x] = deg[y] = d	two 1-bits at bit position d form a **Carry** 1-bit at position d+1
INSERT(x, H) n = |H|	**Increment** n to n+1
UNION(H_1, H_2) n_1 =|H_1| , n_2= |H_2|	**Add** $n_1 + n_2$

Analysis

We will describe heap op's & analyze their worst-case and amortized times.

POTENTIAL: $\Phi(H) = a \cdot t(H) + b \cdot D(H)$

a, b = appropriate non-negative constants to be determined (as we shall see, a = 1 and b = 4 will work fine).

t(H) = # Binomial trees in H (i.e., # roots).

D(H) = maximum node degree in H (i.e., degree of last root).

NOTES:

1. This is a regular potential.

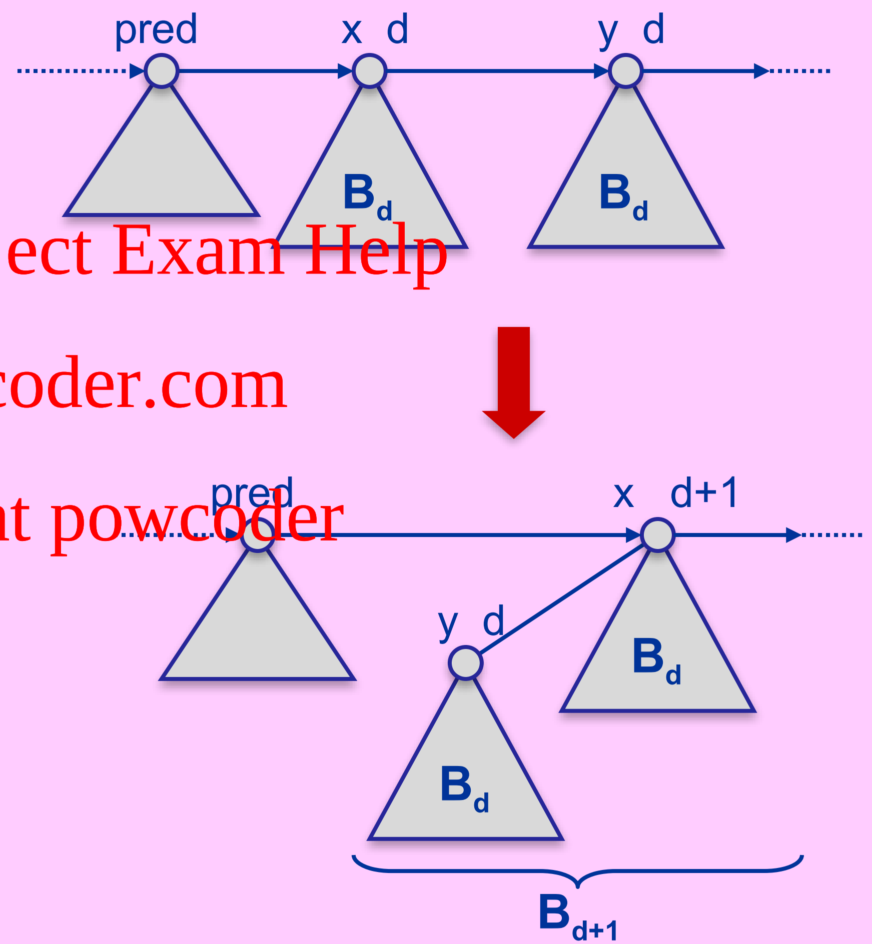
2. $t(H) \leq 1 + D(H) \leq 1 + \log n(H)$.

3. Union merges the two root-lists into one, and repeatedly LINKs equal-degree roots until root degrees are in strictly increasing order.
   The cost of each LINK is paid for by a drop in t(H).
   The cost of merge is paid for by disappearance of the smaller D(H).

LINK

LINK(x,y) (* roots x,y, deg[x] = deg[y] , key[x] ≤ key[y], rsib[x] = y *)

next ← rsib[y]
rsib[y] ← lchild[x]
lchild[x] ← y
p[y] ← x
deg[x] ← deg[x] + 1
rsib[x] ← next
return x
end



Running Time:

c = 1 = O(1).

$\hat{c} = c + \Delta\Phi \le 1 + [-a + b] = O(1).$

t(H) decreases by 1

D(H) may increase by 1
(applies to last LINK only)

MAKEHEAP and FINDMIN

MakeHeap(e)

H ← pointer to a new node x that contains item e and its key key[e]

deg[x] ← 0; rsib[x] ← lchild[x] ← p[x] ← nil

return H

end

Running Time:

c = O(1). ĉ = c + ΔΦ = c + a = O(1).

FindMin(H)

Scan through root-list of H. Find and return min key.

end

Running Time:

c = O(log n). ĉ = c + ΔΦ = c + 0 = O(log n).

UNION

Union(H_1 , H_2)

1. **First scan:** merge root lists of H_1 and H_2 in ascending order of root-degree.
   Stop the merge as soon as one root-list is exhausted.
   Set a pointer last at that spot and append the rest of the other root-list.
   [Now there may be up to 2 roots of any given degree on the merged root-list.]

2. **Second scan:** scan the merged root-list & do "carry operations", i.e., whenever you see 2 or 3 consecutive roots of equal degree, LINK their last 2 (first swap them on the root-list if key of 1st root > key of 2nd root).
   Stop 2nd scan when ∃ no more carry & have reached or passed position last.

3. **return** a head pointer to the head of the merged root-list.

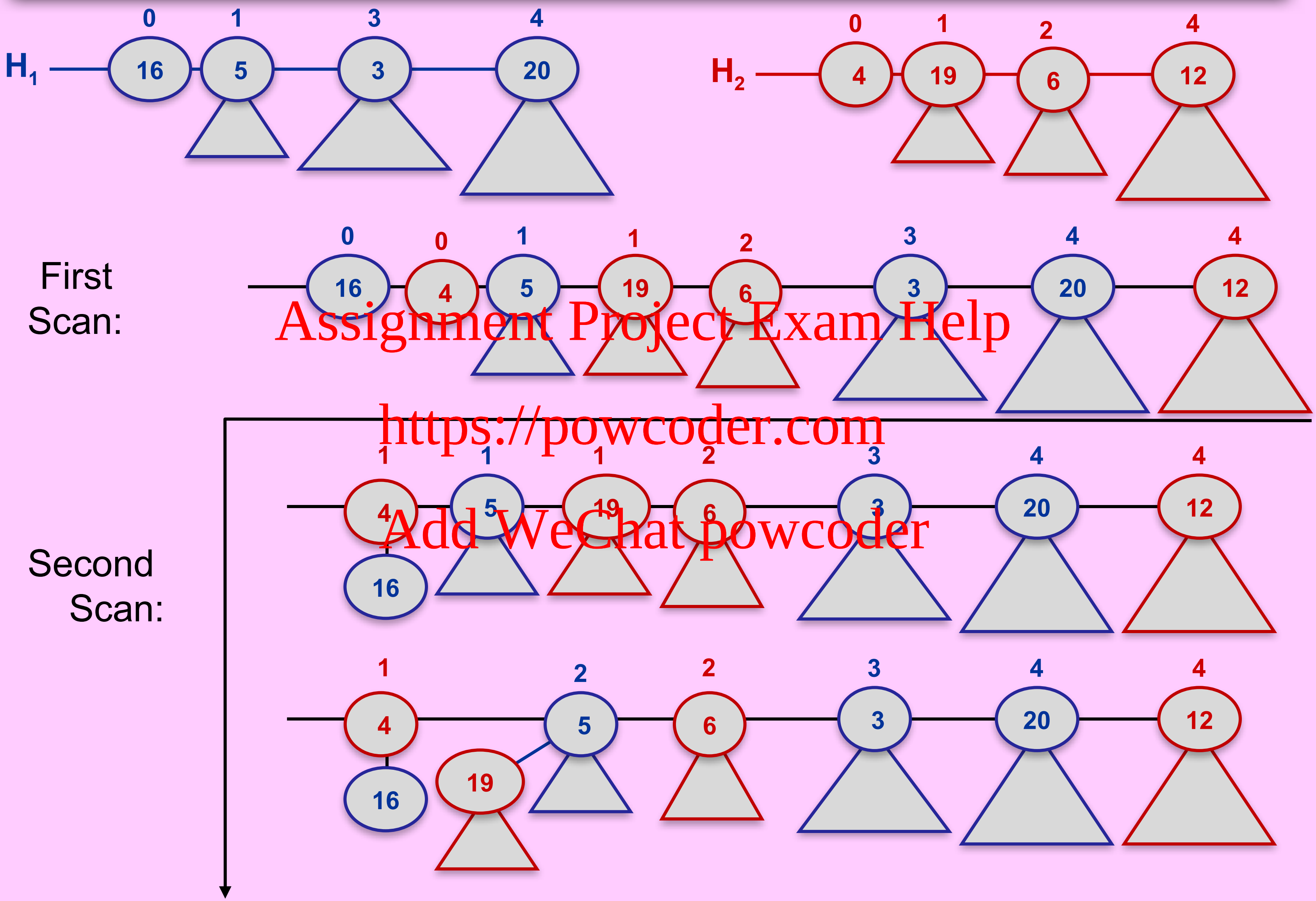
end

Analogy to binary addition:

carry =			1	1			1	1	
n_1 =	1	1	0	1	1	0	0	1	1
n_2 =					1	1	0	1	1
stop 1st scan					↑				
stop 2nd scan			↑						
$n=n_1+n_2$ =	1	1	1	0	0	1	1	1	0

carry =					1		1	
n_1 =	1	0	1	0	0	1	0	1
n_2 =				1	0	1	0	1
stop 1st scan				↑				
stop 2nd scan				↑				
$n=n_1+n_2$ =	1	0	1	1	1	0	1	0

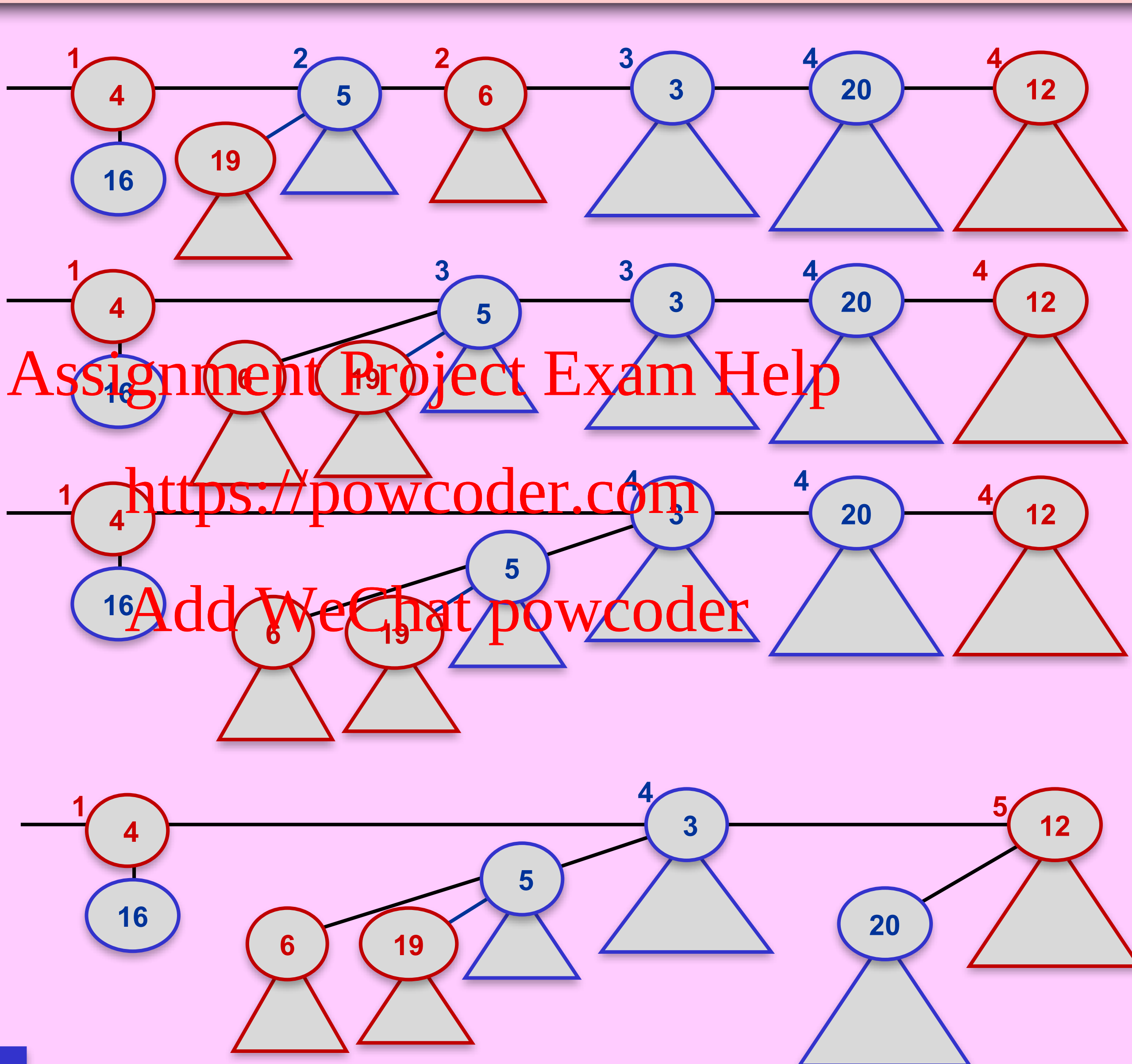
Example 1/3



P.T.O.

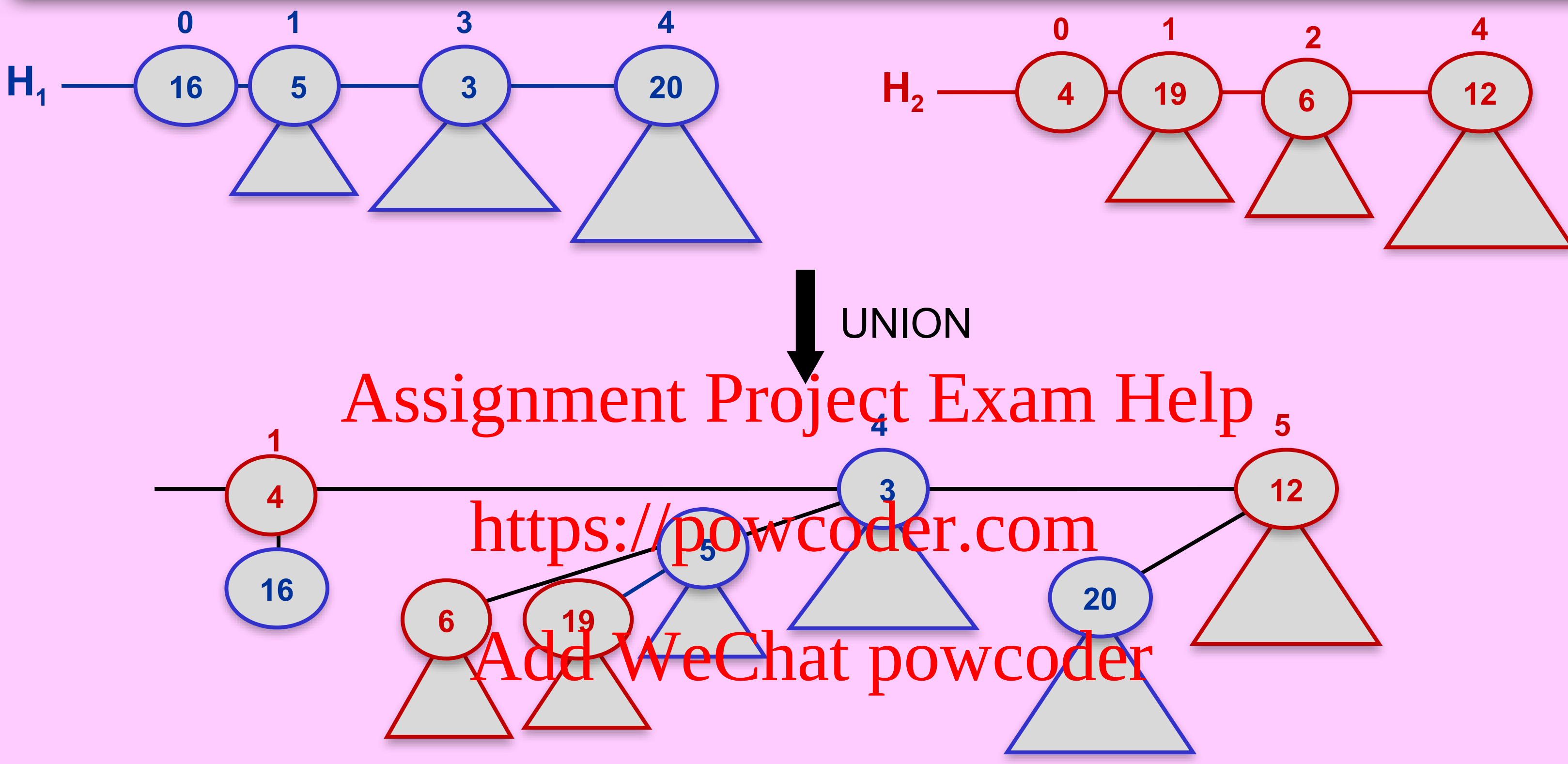
Example 2/3

Second Scan continued:



P.T.O.

Example 3/3



carry =	1	1	1	1	1	
n_1 =		1	1	0	1	1
n_2 =		1	0	1	1	1
$n = n_1 + n_2$ =	1	1	0	0	1	0

UNION Worst-case time

$H = H_1 \cup H_2$

$n_1 = |H_1|$

$n_2 = |H_2|$

$n = n_1 + n_2$

Each of the two scans of the root lists spend O(1) time per root (advancing the scan or LINKing).

$c = O(t(H_1) + t(H_2)) = O((1+\log n_1) + (1 + \log n_2)) = O(\log n)$.

UNION Amortized time

WLOGA: $D(H_1)\quad D(H_2)$

$H = H_1 \cup H_2$

$\Phi(H) = a\,t(H) + b\,D(H)$

$\Phi(H_1) = a\,t(H_1) + b\,D(H_1)$

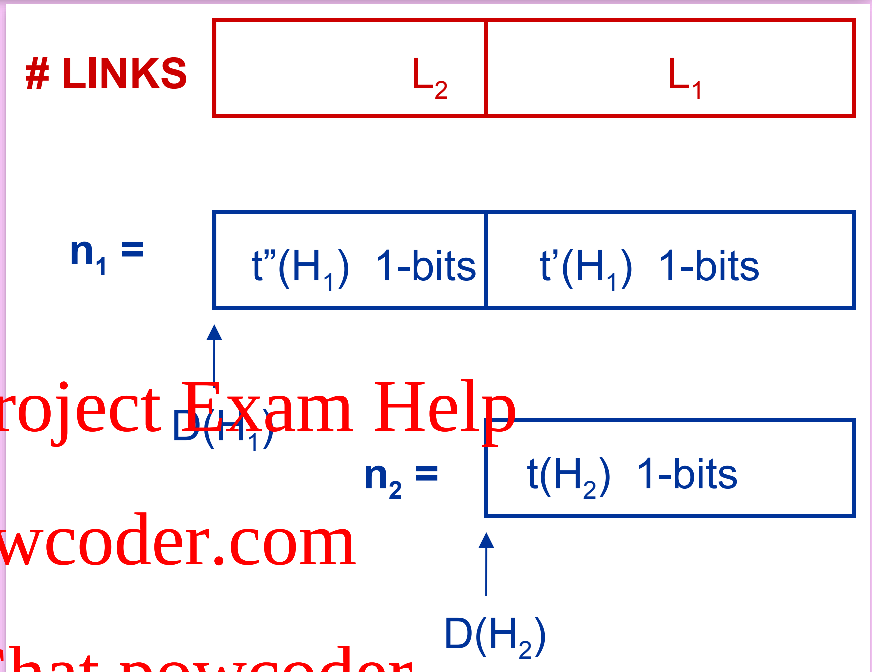
$\Phi(H_2) = a\,t(H_2) + b\,D(H_2)$

$t(H_1) = t'(H_1) + t''(H_1)$

$t(H) = t(H_1) + t(H_2) - L_1 - L_2$

$t'(H_1) + t(H_2) \le 2\,(1 + D(H_2))$

$D(H) \le 1 + D(H_1)$



$c = 1 + c_1 + c_2$ Union cost

$c_1 = t'(H_1) + t(H_2)$ 1st scan

$c_2 = t'(H_1) + t(H_2) + L_1 + L_2$ 2nd scan

$\Delta\Phi = \Phi(H) - \Phi(H_1) - \Phi(H_2) = a[t(H) - t(H_1) - t(H_2)] + b[D(H) - D(H_1) - D(H_2)]$

$\le -a[L_1 + L_2] + b[1 - D(H_2)]$

$\hat{c} = c + \Delta\Phi \le 1 + 2[t'(H_1) + t(H_2)] + (1-a)[L_1 + L_2] + b[1 - D(H_2)]$

$\le 1 + 4[1 + D(H_2)] + (1-a)[L_1 + L_2] + b[1 - D(H_2)]$

$= [5+b] + (1-a)[L_1 + L_2] + (4-b)\,D(H_2)$

$\le [5+b] = O(1)$ if $a\ 1$ and $b\ 4$.

if $a\ 1$ & $b\ 4$,

$\hat{c} = O(1)$.

INSERT

Insert(e,H)

1. H’ ← MakeHeap(e)
2. H ← Union(H,H’)

end

Running Time: (line 1 plus line 2)

$c \;=\; c_1 \;+\; c_2 \;=\; O(1) + O(\log n) = O(\log n).$

$\hat{c}_1 \;=\; c_1 \;+\; \Delta\Phi_1 \;=\; O(1).$

$\hat{c}_2 \;=\; c_2 \;+\; \Delta\Phi_2 \;=\; O(1).$

$\Delta\Phi = \Delta\Phi_1 + \Delta\Phi_2 .$

$\hat{c} \;=\; c \;+\; \Delta\Phi \;=\; \hat{c}_1 + \hat{c}_2 = O(1).$

DELETEMIN

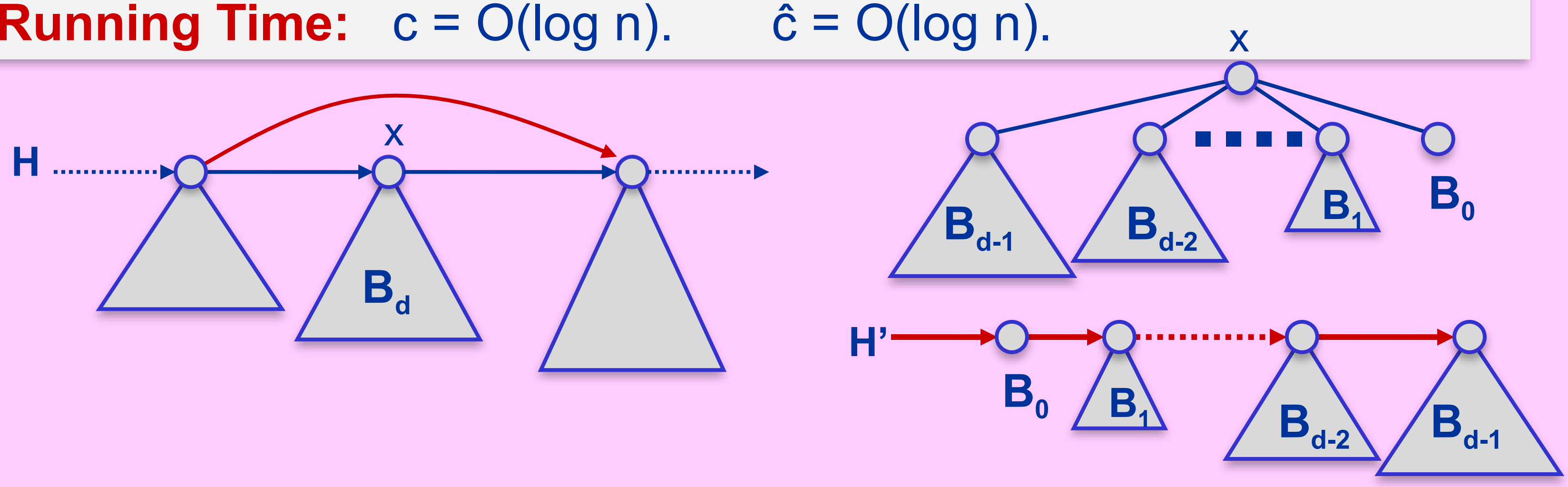
DeleteMin(H) (* assume H $\neq \varnothing$ *)

1. $x \leftarrow$ Minimum key root in H, found via scan of the root-list
2. & x detached from root-list of H
3. MinKey $\leftarrow$ key[x]
4. Scan-&-modify child-list of x:
   H' $\leftarrow$ head pointer to child-list of x reversed (min-to-max degree), Assignment Project Exam Help
   p[c] $\leftarrow$ nil, for each child c of x.
5. H $\leftarrow$ Union(H,H') https://powcoder.com
6. **return** MinKey

end

Add WeChat powcoder

Running Time: $c = O(\log n)$. $\hat{c} = O(\log n)$.



DECREASEKEY & DELETE

```
DecreaseKey(x, K, H)        (*percolate item at node x up its tree*)
    if key[x] < K then return error
    key[x] ← K
    while p[x] ≠ nil and key[x] < key[p[x]] do
                key[x] ↔ key[p[x]]
                x ← p[x]
end
```


Running Time: $c = O(\log n)$. $\hat{c} = c + \Delta\Phi = c + 0 = O(\log n)$.


```
Delete(x,H)
    DecreaseKey(x, -∞, H)
    DeleteMin(H)
end
```

Running Time: $c = O(\log n)$. $\hat{c} = O(\log n)$.

Binomial Heap Complexity

Operation	**Worst-case**	**Amortized**
MakeHeap	O(1)	O(1)
FindMin	O(log n) * O(1)	O(log n) * O(1)
Union	O(log n)	O(1)
Insert	O(log n)	O(1)
DeleteMin	O(log n)	O(log n)
DecreaseKey	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Fibonacci Heap (discussed next) improves the amortized times.

Fibonacci Heaps

Fibonacci vs Binomial Heaps

- **Trade off:** Fibonacci Heaps were designed by Fredman and Tarjan [1987], primarily to improve efficiency of network (graph) optimization problems. In such applications, DecreaseKey is used more frequently than DeleteMin (# edges vs # vertices in the graph). Make DecreaseKey cheaper at the possible expense of DeleteMin.

- Fibonacci Heaps may be considered as a quasi self-adjusting version of Binomial Heaps with the following main alterations:

- **Alteration 1:** Union and Insert are done by lazy-merge of the two root-lists.

- **Alteration 2:** A node in Fib-Heap can have large depth (up to O(n)). Instead of percolating, DecreaseKey is done by a controlled link cutting. The control is done by a node marking scheme. Delete invokes DecreaseKey as in Binomial Heaps.

  [If DecreaseKey and Delete are never used, then no CUT takes place, and as a result, Fibonacci trees would be (unordered versions of) Binomial trees.]

- **Alteration 3:** While scanning the root-list to find the new min key, DeleteMin cleans up (consolidates) the messy root-list caused by repeated cuts and lazy-merges.

- **Result:** O(1) amortized time for MakeHeap, FindMin, Union, Insert, DecreaseKey. O(log n) amortized time for DeleteMin and Delete.

Fibonacci Heaps

- A Fibonacci Heap H is an unordered forest of node-disjoint heap-ordered trees. (Root-degrees, not necessarily distinct, appear in arbitrary order on root-list.)

- The root-list, as well as each child-list, is a doubly-linked-circular-list (DLCL). This is done by right-sibling and left-sibling pointers at each node.

- In O(1) time we can concatenate 2 DLCLs, & add or remove a node from such a list.

- **min[H]** = pointer to min-key root (this is the entry point to the DLCL root-list of H).

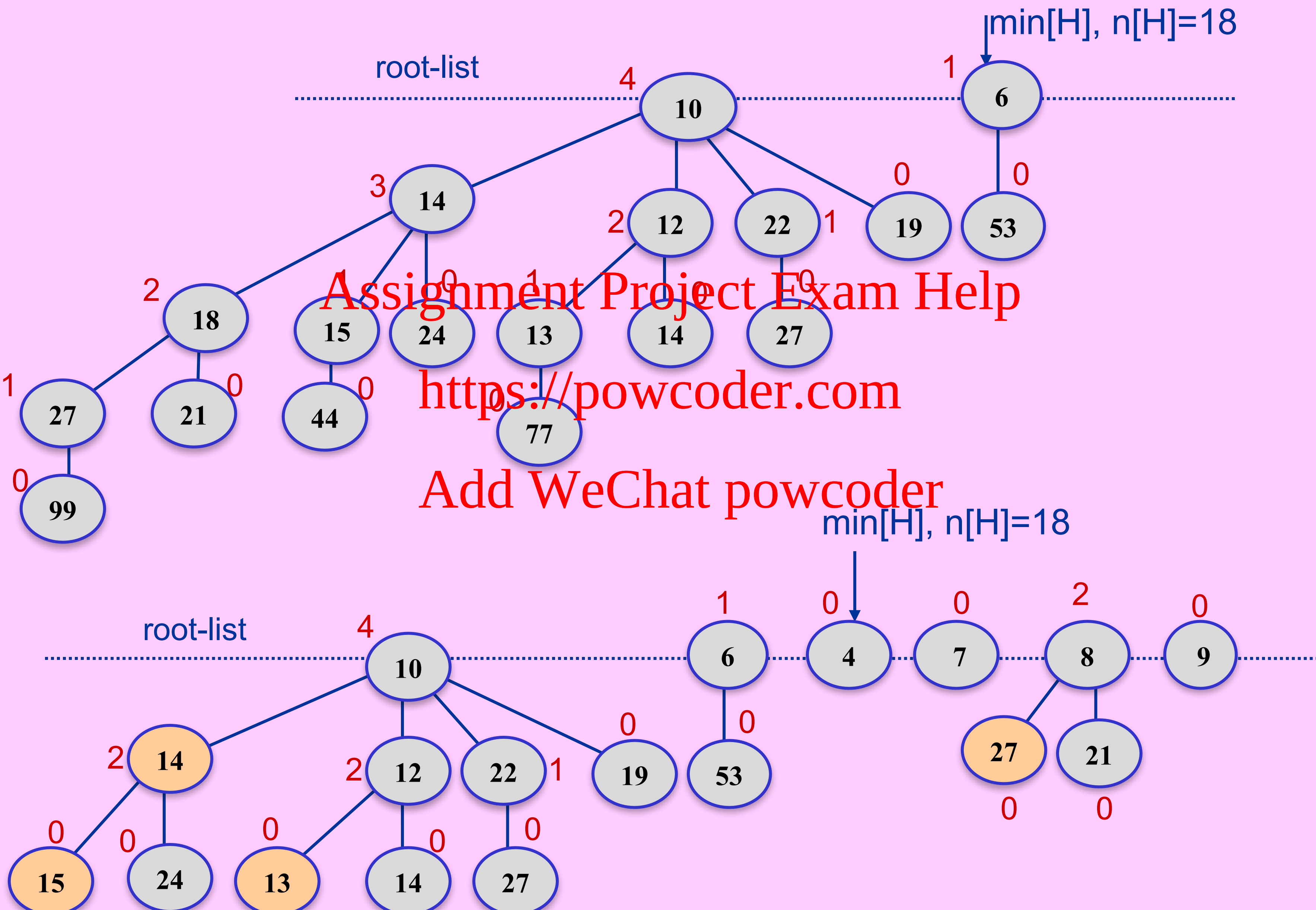
  n[H] = # items in H.

- **Node x structure:** key[x], deg[x], rsib[x], lsib[x], p[x], child[x], mark[x].

child[x] = pointer to an arbitrary child of x.

mark[x] =
- true — if x lost a child (by CUT) since the last time p[x] was updated (by a LINK or CUT or CONSOLIDATE)
- false — otherwise

Example



min[H], n[H]=18
root-list
10
6
14
12
22
19
53
18
15
24
13
14
27
27
21
44
77
99
min[H], n[H]=18
root-list
10
6
4
7
8
9
14
12
22
19
53
27
21
15
24
13
14
27

Analysis

We will describe heap operations & analyze their amortized times.

POTENTIAL: $\Phi(H) = 2t(H) + 3m(H)$

$t(H)$ = # trees in H (i.e., # roots),
$m(H)$ = # marked nodes in H.

Brief explanation:

the O(1) cost of each LINK or CUT operation is paid for by a corresponding drop in potential.

- Each LINK reduces t(H) by 1 and does not increase m(H).
- Each CUT increases t(H) by 1, but in a cascading series of CUTs of (marked) nodes, each CUT (except possibly the first and the last) reduces m(H) by 1 also.

D(n)

An **Unordered Binomial tree** of order d, denoted $\mathbf{U_d}$:

- $U_0 = B_0$.
- For d>0, $\mathbf{U_d}$ has a root of degree d whose d subtrees are $U_0, U_1, U_2, \ldots, U_{d-1}$ in some arbitrary order.
- Items in $\mathbf{U_d}$ are heap-ordered.

FACT:

- U_d has 2^d nodes and maximum node degree d.
- LINK joins two equal degree roots (and maintains heap-order).
- CUT is invoked by DecreaseKey and Delete only.
- If no CUT is done, i.e., only mergeable heap operations, then every tree in Fibonacci Heap is U_d, for some d.

Max-Degree Claim:

D(n) := maximum degree of any node in any n-node Fibonacci Heap.

- D(n) = O(log n). (Proved later.)
- $D(n) = \lfloor \log n \rfloor$, if only mergeable heap operations are performed.

LINK

LINK(x,y,H)

move y next to x on the root-list DLCL

if key[x] > key[y] **then** x ↔ y

remove y from root-list DLCL

insert y in child-list DLCL of x

p[y] ← x

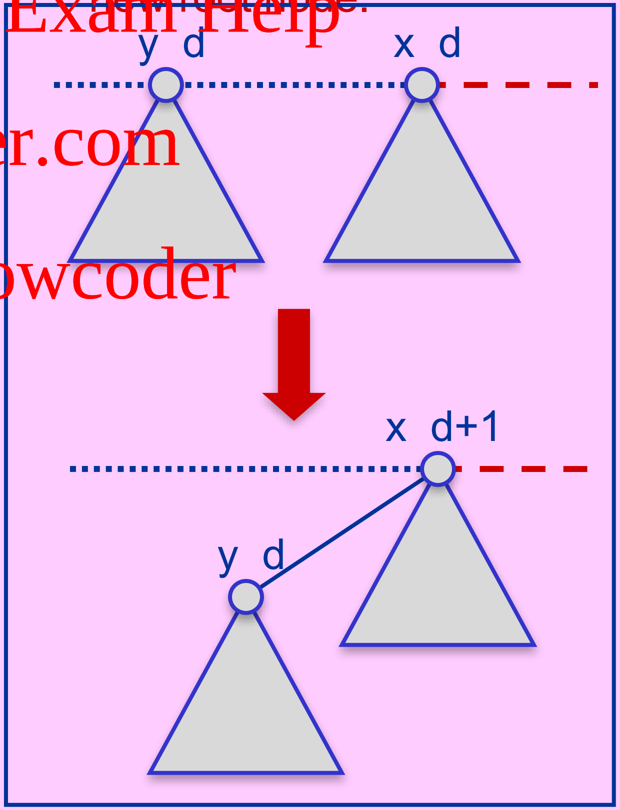
mark[y] ← false

deg[x] ← deg[x] + 1

end

Pre-Cond: deg[x]=deg[y], roots x,y appear anywhere on root-list, not necessarily consecutive.

Post-Cond: LINK x,y at position x in root-list and make x point to new root node.



Running Time:

$\Phi(H) = 2t(H) + 3m(H)$

$\hat{c} = c + \Delta\Phi \le 1 + [-2] = O(1).$

t(H) decreases by one
m(H) doesn't increase

MAKEHEAP and FINDMIN

MakeHeap(e)

```
        n[H] ← 1
    min[H] ← pointer to a new node x containing key[e]
    deg[x] ← 0
        mark[x] ← false
        rsib[x] ← lsib[x] ← x    child[x] ← p[x] ← nil
    return  H
end
```


FindMin(H)

```
        if min[H]=nil  then
return nil
        return key[min[H]]
end
```

Running Time: $\hat{c} = O(1)$.

UNION and INSERT

Union(H_1, H_2)

if $\min[H_1] = \text{nil}$ **then return** H_2

if $\min[H_2] = \text{nil}$ **then return** H_1

concatenate DLCL root-lists of H_1 and H_2

$n[H_2] \leftarrow n[H_1] + n[H_2]$

if $key[\min[H_1]] < key[\min[H_2]]$ **then** $\min[H_2] \leftarrow \min[H_1]$

return H_2

end

Insert(e, H)

$H' \leftarrow \text{MakeHeap}(e)$

$H \leftarrow \text{Union}(H, H')$

end

Running Time: $\hat{c} = c + \Delta\Phi \le c + 2 = O(1)$.

DELETEMIN

```
DeleteMin(H)                                                    (*assume H ≠ ∅*)
1.  z ← min[H];   MinKey ← key[z]
2.  modify root-list of H pointed by min[H]:   remove z from it, and instead
    concatenate child-list of z (pointed by child[z]) to it.     (*now min[H] ≠ z*)
3.  n[H] ← n[H] – 1
4.  if n[H] = 0 then min[H] ← nil
5.                   else Consolidate(H)       (*update min[H] & clean-up root-
  list*)
6.  return MinKey
end
```



```
Consolidate(H)
7.   for d ← 0 .. D(n[H]) do A[d] ← nil       (*degree-indexed root pointer table*)
8.   for each root x on the root-list of H do                   (*scan root-list*)
9.       if key[x] < key[min[H]] then min[H] ← x                (*update min[H]*)
10.      p[x] ← nil;   mark[x] ← false          (*children of old z have new parent*)
11.      d ← deg[x]
12.      while A[d] ≠ nil do                      (*LINK repeatedly till deg[x] *)
13.          LINK(x, A[d], H);  A[d] ← nil; d ← d+1  (* ≠ deg of scanned roots *)
14.      end-while
15.      A[d] ← x
16.  end-for
end
```


Cost: c_1: 1-7 (ex 5), c_2: 12-14 (over all 8-16), c_3: 8-16 (ex 12-14).

DELETEMIN Amortized time

Proof idea: Cost of while-loop is paid for by the LINKs.
The rest charges O(1) per root of distinct degree (at most 1+D(n)).

Proof detail:

H = the initial heap
H' = heap just before call to Consolidate
H'' = the final heap
L = total # LINK operations performed by Consolidate

NOTE: $t(H'') = t(H) + [deg[z] - 1 - L] = t(H') - L \le 1 + D(n)$

$c_1 = 1 + D(n)$ .... cost of lines:1-7 (excluding 5)

$c_2 = L$ .... total cost of the while-loop:12-14 (over all for-loop iterations:8-16)

$c_3 = t(H') \le 1 + D(n) + L$ ... cost of the for-loop:8-16, excluding the while-loop:12-14

$$c = c_1 + c_2 + c_3 \le 2 + 2D(n) + 2L$$

$$\begin{aligned} \Delta\Phi &= 2\,[t(H'') - t(H)] + 3\,[m(H'') - m(H)] \\ &\le 2\,[t(H'') - t(H)] = 2\,[deg[z] - 1 - L] \le 2D(n) - 2 - 2L \end{aligned}$$

$$\begin{aligned} \hat{c} &= c + \Delta\Phi \le 4\,D(n) \\ &= O(\log n) \qquad \text{[by the Max-Degree Claim]} \end{aligned}$$

CUT, DECREASEKEY, DELETE

```
CUT(x,H)        (* unlink x from its non-nil parent *)
     y ← p[x]
     remove x from child-list of y and add x to root-list of H
     p[x] ← nil ;   mark[x] ← false
     deg[y] ← deg[y] – 1
end
```

$\hat{c} = c + \Delta\Phi \le 1 + 2 = O(1)$

```
DecreaseKey(x, K, H)   (* K ≤ key[x] *)
     key[x] ← K;  y ← p[x]
     if  y ≠ nil  &  K < key[y]  then do
          CUT(x, H)
          while  p[y] ≠ nil  and  mark[y]  do
                z ← p[y]; CUT(y, H); y ← z
          end-while
          mark[y] ← true
     end-if
     if  K < key[min[H]]  then  min[H] ← x
end
```

$\hat{c} = O(1)$

$\hat{c} = c + \Delta\Phi = 1 + [2 – 3] = 0$

$\hat{c} = O(1)$

$\hat{c} = O(1)$

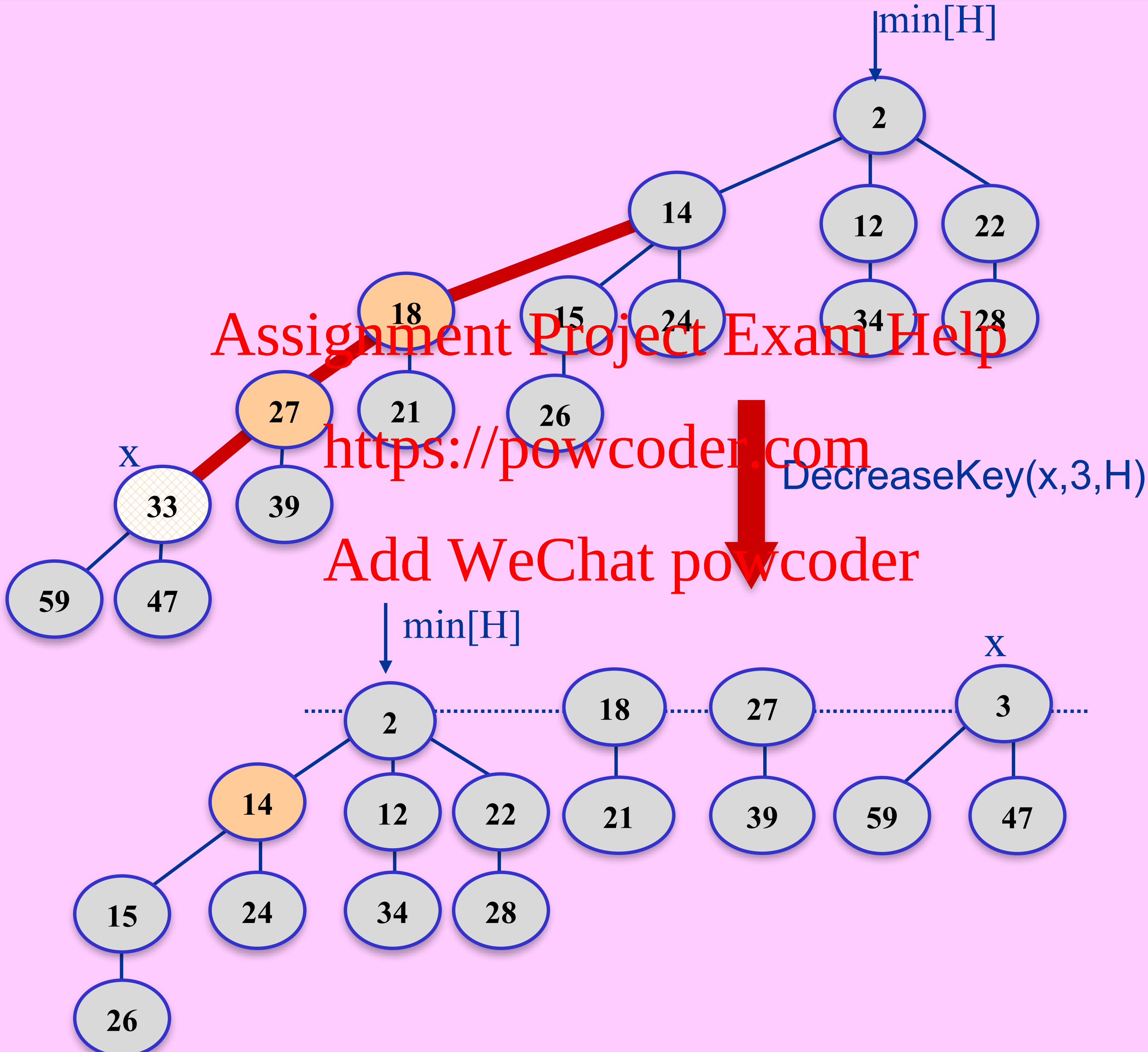
```
Delete(x,H)
     DecreaseKey(x, -∞, H)
     DeleteMin(H)
end
```

$\hat{c} = O(1)$

$\hat{c} = O(\log n)$

$\hat{c} = O(\log n)$

DecreaseKey Example



Max-Degree Claim

Max-Degree Claim:

D(n) := maximum degree of any node in any n-node Fibonacci Heap.

- D(n) = O(log n).
- D(n) = ⌊log n⌋ , if only mergeable heap operations are performed.

- We need to prove the bound D(n) = O(log n) which was used in the amortized analysis of DeleteMin.

- We also need to derive an explicit upper-bound formula for D(n) needed to allocate the array A[0..D(n)] in procedure Consolidate. We will do so shortly.

  - Note: there is a way around this by making array A a dynamic table with O(1) amortized cost per table expansion or contraction! Some subtlety is involved. Fill in the details; how?

Fibonacci Numbers

Fibonacci Numbers:

F(0) =0, F(1)=1, F(d+2) = F(d+1) + F(d), for all d 0.

d:	0	1	2	3	4	5	6	7	8	9	10	11	12	…
F(d):	0	1	1	2	3	5	8	13	21	34	55	89	144	…

FACT: For all d 0 we have:

(1) $F(d+2) = 1 + F(0) + F(1) + \cdots + F(d)$

(2) $F(d+2) \quad \varphi^d \quad \left(\text{golden ratio: } \varphi = \frac{1+\sqrt{5}}{2} = 1.61803\cdots, \quad \varphi^2 = \varphi + 1.\right)$

Proof: By induction on d:

(1) Base (d=0,1): F(2) = 1 = 1+F(0), F(3) = 2 = 1+F(0)+F(1).

Ind. Step (d>1): F(d+2) = F(d+1) + F(d) = [1+F(0)+ … +F(d-1)] + F(d).

(2) Base (d=0,1): F(2) = 1 = φ^0, F(3)=2 φ^1.

Ind. Step (d>1): F(d+2) = F(d+1) + F(d)

$\varphi^{d-1} + \varphi^{d-2} = \varphi^{d-2}(\varphi + 1) = \varphi^{d-2}\,\varphi^2 = \varphi^d$.

Fibonacci Heap node degrees

LEMMA 1: Suppose deg[x]=d, and children of x are $y_1, y_2, \ldots, y_d$, in the order they were LINKed under x (from oldest to youngest). Then, $deg[y_i]$ max { 0, i – 2 } , for i=1..d.

Proof: When y_i was about to be LINKed under x, nodes $y_1, y_2, \ldots, y_{i-1}$ (and possibly other nodes) were already children of x.

So at that time, $deg[y_i] = deg[x]$ i – 1 .

CLAIM: y_i could have lost at most 1 child since then.

Why? Because y_i has remained a child of x. If during this time period y_i lost a 1st child (due to a CUT), y_i would become & remain marked. [Since y_i is not a root, it could not have subsequently become unmarked by Consolidate. Also, if this marked y_i would have lost a 2nd child (due to a CUT), it would be unmarked, but y_i would be cut from x and not remain a child of x. Even if later on y_i became a child of x again (due to a LINK), it would no longer be a child of x in the given chronological order.]

So, now $deg[y_i]$ i – 2 .

Fibonacci Heap node sizes

LEMMA 2: For all nodes x, size(x) $\varphi^{deg[x]}$,
where size(x) denotes the # descendants of x, inclusive.

Proof: s(d) := minimum possible size of any degree d node in a Fib-H.

CLAIM: s(d) F(d+2). [Fact (2): F(d+2) φ^d completes the proof.]

Proof of Claim by induction on d:

Base (d=0,1,2): s(0)=1=F(2), s(1) 2=F(3), s(2) 3=F(4).

Ind. Step (d>2): s(d) = size(x) for some node x, s.t. deg[x]=d, with d children (from oldest to youngest) $y_1, y_2, \ldots, y_d$.

(a) size(x) = 1 + size(y_1) + size(y_2) + … + size(y_d).

(b) size(y_1) 1 = F(0) + F(1).

(c) By Lemma 1, deg(y_i) i – 2 , i = 2..d.

(d) By induction hypothesis, size(y_i) s(i – 2) F(i), for i = 2..d.

(e) Thus, s(d) = size(x) 1 + F(0) + F(1) + F(2) + … + F(d) = F(d+2).

Max-Degree Claim

Max-Degree Claim: D(n) = O(log n).

Proof: By Lemma 2:

$n \;\; size(x) \;\; \varphi^{deg[x]}$, $\forall$node x in any n-node Fibonacci Heap.

$\therefore\ deg[x] \le \log_{\varphi} n \quad (< 1.441 \log n)$.

Define: $\mathbf{D(n)} = \lfloor \log_{\varphi} n \rfloor$.

Fibonacci Heap Complexity

Operation	Worst-case	Amortized
MakeHeap	O(1)	O(1)
FindMin	O(1)	O(1)
Union	O(1)	O(1)
Insert	O(1)	O(1)
DeleteMin	O(n)	O(log n)
DecreaseKey	O(n)	O(1)
Delete	O(n)	O(log n)

Brodal Heap (mentioned next) turns these amortized times into worst-case.

Recent Developments

Other Priority Queues

- Soft Heap by Chazelle [2000]
  - Approximate Binomial heaps: up to εn items are *corrupted*.
  - Amortized times: O(log 1/ε) for Insert, O(1) for all other operations.
  - Approximate sorting with up to εn errors.
  - O(n) time exact median-finding and selection.
  - O(m α(m,n)) time MST algorithm.

- Simplified Soft Heap by Kaplan and Zwick [2009]

- Strict Fibonacci Heaps by Brodal, Lagogiannis and Tarjan [2012]
  - Worst-case times match amortized times in Fibonacci Heaps.

- Hollow Heaps by Hansen, Kaplan, Tarjan, Zwick [2017]
  - Much simpler and as efficient as Fibonacci Heaps.
  - Uses DAG instead of forest-of-trees structure, and lazy DecreaseKey that creates hollow nodes.

- References on the next page.

- AAW animations: Leftist, Skew, Binomial, and Fibonacci Heaps.

Bibliography:

C.A. Crane [1971] also described in [D.E. Knuth, "Art of Computer Programming," vol. 3:151-152, 1973.] [Leftist Heaps]

Sleator, Tarjan, "Self-adjusting heaps," SIAM J. Computing 15(1):52-69, 1986. [Skew Heaps]

M.R. Brown, "The analysis of a practical and nearly optimal priority queue," PhD thesis, CS Dept., Stanford University, Technical Report STAN-CS-77-600, 1977. [Binomial Heaps]

M.R. Brown, "Implementation and analysis of binomial queue algorithms," SIAM J. Computing, 7(3):298-319, 1978.

J. Vuillemin, "A data structure for manipulating priority queues," Communication of the ACM, 21(4):309-315, 1978. [Binomial Heaps]

Fredman, Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," JACM, 34(3):596-615, 1987.

B. Chazelle, "The Soft Heap: an approximate priority queue with optimal error rate," JACM, 47(6):1012-1027, 2000.

B. Chazelle, "A minimum spanning tree algorithm with inverse Ackermann type complexity," JACM, 47(6):1028-1047, 2000.

H. Kaplan, U. Zwick, "A simpler implementation and analysis of Chazelle's Soft Heaps," SODA, pp:477-485, 2009.

B. Haeupler, S. Sen, R.E. Tarjan "Rank-Pairing Heaps," SIAM J. Computing 40(6):1463-1485, 2011.

G.S. Brodal, G. Lagogiannis, R.E. Tarjan, "Strict Fibonacci Heaps", Symposium on Theory of Computing (STOC): 1177-1184, 2012.

T.D. Hansen, H. Kaplan, R.E. Tarjan, U. Zwick "Hollow Heaps", ACM Transactions on Algorithms, Vol. 13, No.3. Article 42, July 2017.

Exercises

1. **Leftist and Skew Heap example sequence:**
   (a) Perform the sequence of leftist-heap operations Insert(36), DeleteMin, Insert(17), starting with the Leftist Heap on the bottom of Figure 2 of Lecture Note 4. Show the result after each operation. Also show the dist values of the nodes.
   (b) Now perform the same sequence as Skew Heap operations on the same initial tree interpreted as a Skew Heap.

2. **Union on Skew Heap:**
   (a) Write a recursive procedure for the Union operation on Skew Heaps. Stop the recursion as soon as one rightmost path is exhausted. Redo the amortized analysis. Any change to the amortized time?
   (b) Now do the same with a top-down (one-pass) iterative Union on Skew Heaps.

3. **DecreaseKey and Delete on Leftist & Skew Heaps:**
   (a) Show how to perform DecreaseKey and Delete on a Skew Heap in O(log n) amortized time each, using the same potential function as before. (You may assume each node has a parent pointer.)
   (b) Show how to perform DecreaseKey and Delete on Leftist Heaps that take O(log n) worst-case time each. Prove your time bounds.

4. **Worst-case Skew Heap Union:** We claim the amortized time bound of O(log n) for the Skew Heap operations does not apply as worst-case bound. Show this by giving a sequence of O(n) mergeable-heap operations, starting with no heaps, that leads to a Union operation requiring $\Theta(n)$ actual time.

5. **Binomial heap batch insertion:** Design and analyze a simple algorithm to insert m keys, given in arbitrary order, into a Binomial Heap of size n in O(m + log n) worst-case time.

6. **Lazy Delete in Leftist Heaps:** One way to delete nodes from a known position in a leftist heap is to use a lazy strategy. To delete a node, merely mark it deleted (this requires an additional boolean field in each node). When a FindMin or DeleteMin is performed, there is a potential problem if the root is marked deleted, since then the node has to be actually deleted and the real minimum needs to be found, which may involve deleting other marked nodes. In this strategy, each Delete costs one unit of time, but the cost of a DeleteMin or FindMin depends on the number of nodes that are marked deleted. Suppose that after a DeleteMin or FindMin there are R fewer marked nodes than before the operation.

(a) Show how to perform the DeleteMin in O(R log n) time.

(b) Propose an implementation, with an analysis to show that the time to perform DeleteMin (on leftist heaps with lazy deletion) can be improved to O(R log (2n/R)).

7. **Construct Heap:** The algorithm below returns a heap of a set S of n arbitrary given keys.

algorithm ConstructHeap(S)

1. place each element of S into a size 1 heap by itself
2. place (a pointer to each of) these |S| heaps in an initially empty queue Q

3a. **while** $|Q| > 1$ **do**

3b. dequeue two heaps H_1 and H_2 from the front of Q

3c. enqueue the heap Union(H_1 , H_2) at the rear of Q

3d. **end-while**

4. $H \leftarrow$ dequeue(Q)
5. **return** H

end

(a) Show the output of ConstructHeap({23, 14, 3, 86, 18, 35}) using Leftist Heaps.

(b) Answer the same question (a) for Binomial Heaps instead.

(c) Show that ConstructHeap applied to Skew Heaps takes O(n) time in the worst case.

8. **DecreaseAllKeys:** Suppose we want to add the DecreaseAllKeys(D, H) operation to the heap repertoire. The result of this operation is that all keys in heap H have their value decreased by an amount D. For the heap implementation of your choice, explain the necessary modifications so that all other operations retain their asymptotic running times & DecreaseAllKeys runs in O(1) time.

9. **Comparing various heaps:** Let s be a sequence of mergeable-heap operations applied to initially no heaps. For each case below demonstrate one such sequence s with the stated property, or argue why it cannot exist.
(a) s takes) time on Skew and Leftist Heaps, but time on Binomial Heaps.
(b) s takes) time on Binomial Heaps, but time on Skew Heaps.
(c) s takes) time on Skew Heaps, but time on Leftist Heaps.
(d) s takes) time on Leftist Heaps, but time on Skew Heaps.

10. **Improved Binomial heap insertion:** We know that in a Binomial Heap the roots on the root-list appear in strictly increasing order of degree. This obviously implies that no two roots have equal degree. Now suppose we relax this condition and instead require that no three roots have equal degree. Can we obtain O(1) worst-case time for insertion without degrading the worst-case time complexities of the other heap operations? Explain.
[Hint: maintain a suitable sparseness among pairs of roots of equal degree.]

11. **[CLRS, Exercise 19.4-1, page 526] Linear Height Fibonacci Heap:**
Every node of an n-node Binomial Heap has depth O(log n). On the contrary, an n-node Fibonacci Heap could have a node of depth Show this be demonstrating a sequence of only O(n) Fibonacci Heap operations that starts with no heaps, and creates a Fibonacci Heap consisting of just one tree that is a linear chain of n nodes.
[Note: Exponentially many operations is not allowed. Each operation must be one of the 7 we defined: MakeHeap, FindMin, DeleteMin, Insert, Union, DecreaseKey, Delete.]

12. **[CLRS, Problem 19-3, page 529] More Fibonacci Heap operations:**
We wish to augment a Fibonacci Heap H to support two new operations without degrading the amortized time of any previously defined Fibonacci Heap operations.

(a) The operation **ChangeKey(x, K, H)** changes the key of node x to the value K. Give an efficient implementation of ChangeKey, and analyze its amortized running time for the cases in which K is greater than, less than, or equal to key[x].

(b) Give an efficient implementation of **Prune(H,r)**, which deletes min{r, n[H]} nodes from H. Which nodes are deleted should be arbitrary and is up to the algorithm to choose. Analyze the amortized running time of your implementation.
[Hint: you may need to modify the data structure and the potential function.]

13. **Modified Fibonacci Heap:** In a Fibonacci Heap H, we explicitly maintain n[H], the number of items in H, and use this to compute D(n). This is required by Consolidate to allocate array A[0..D(n)].

(a) One way to avoid storing n[H] and computing D(n) is to implement array A as a dynamic table with expansion and contraction. Fully explain how this can be done with no adverse effect on the amortized times of any of the Fibonacci Heap operations.

Another modification to Fibonacci Heaps: DecreaseKey allows λ=2 children of a node to be cut from it before it cuts the link between the node and its parent.

(c) What essential quantities would be affected by varying λ (= 2,3,4,…)?
(d) Does using λ =3 affect asymptotic amortized running times of heap operations? Why?
(e) Explain the effect of letting λ to take asymptotically larger values (approaching n).

14. [CLRS 2nd edition, Problem 19-2, page 474] MST algorithm with Priority Queues: Below we consider an alternative to Prim's and Kruskal's MST algorithms. We are given a connected, undirected graph $G=(V,E)$ with a weight function $w: E \rightarrow \Re$. We call $w(u,v)$ the weight of edge (u,v). We wish to find an MST for G: an acyclic subset $T \subseteq E$ that connects all the vertices in V and whose total weight $w(T) = \Sigma_{(u,v)\in T}\, w(u,v)$ is minimized. The MST(G) procedure shown below correctly constructs an MST T. It maintains a vertex-partition $\{V_i\}$ of V, and for each V_i, the set of edges $E_i = \{(u,v) : u \in V_i \text{ or } v \in V_i\}$ incident on vertices in V_i.

algorithm MST(G)
1. $T \leftarrow \emptyset$
2. **for** each vertex $v_i \in V[G]$ **do** $V_i \leftarrow \{v_i\}$; $E_i \leftarrow \{(v_i, v) \in E[G]\}$ **end-for**
3. **while** there is more than one set V_i **do**
4. choose any set V_i
5. delete the minimum weight edge (u,v) from E_i
6. assume without loss of generality that $u \in V_i$ and $v \in V_j$
7. **if** $i \neq j$ **then do** $T \leftarrow T \cup \{(u,v)\}$
8. $V_i \leftarrow V_i \cup V_j$, destroying V_j
9. $E_i \leftarrow E_i \cup E_j$
10. **end-if**
11. **end-while**

end

(a) Design and analyze an implementation of this algorithm that uses Binomial Heaps to manage the vertex and edge sets. [Disjoint Set Union is our next topic of discussion. If you need to use such a data structure as well, properly explain where and how.]

(b) Do the same as in part (a), but use Fibonacci Heaps instead.

(c) [Extra Credit and Optional:] To improve efficiency, we change line 4 as follows: select a smallest cardinality set V_i . Give an efficient implementation of this variant. What is the running time? Did you get any improvement?

15. Search-Heap Tree: We are given a set P = { p_1 , p_2 , … , p_n } of n points in the plane. Assume each point is given by its x and y coordinates, $p_i = (x_i , y_i)$, for i=1..n.

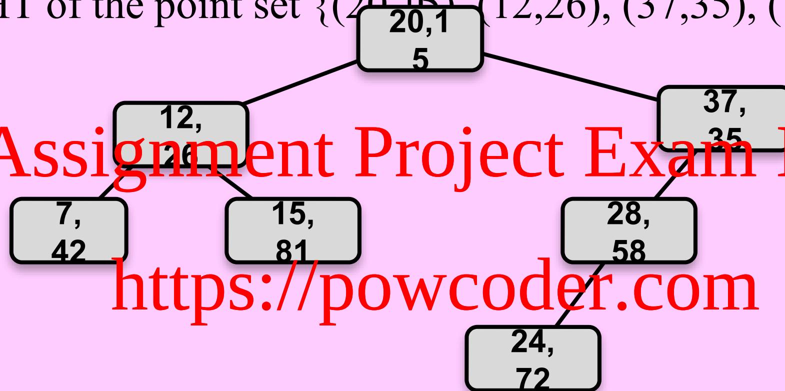
A **Search-Heap Tree (SHT)** of P is an n-node binary tree T with the following properties:

(i) Each node of T holds a distinct point of P,

(ii) T appears as a Binary Search Tree with respect to x-coordinates of its stored points,

(iii) T appears as a min-heap with respect to y-coordinates of its stored points.

Below is an SHT of the point set {(20,15), (12,26), (37,35), (7,42), (15,81), (28,58), (24,72)}.



(a) Show SHT of the point set P = {(12,31), (24,15), (4,23), (18,5), (14,53), (16,7)}.

(b) In general, if all points in P have distinct x coordinates and distinct y coordinates, show that SHT of P exists and is unique. What happens to the existence or uniqueness of SHT if we remove the coordinate distinctness assumption?

(c) Let T represent the SHT of the point set P. Suppose the y-coordinate of a point of P stored at a given node v of T is decreased to y_{new}. How would you update this y-coordinate and use rotation to restore the SHT property of T?

(d) The problem is to construct the SHT of P, where P is a set of n points in the plane, given in sorted order of their x-coordinates. Design and analysis the most efficient algorithm you can for this problem. [Hint: Use incremental construction and amortized analysis.]

16. Min-Max Heaps: In this exercise we want to study a data structure called a **min-max heap**. This data structure supports efficient implementations of both DeleteMax and DeleteMin (as well as Insert). (See an example min-max heap in the figure below.)

A min-max heap is an arbitrary rooted tree with one key per node that satisfies the following min-max heap properties.

(i) All nodes at even depths (the root is at depth zero) are min nodes, and all nodes at odd depths are max nodes. That is, the root is a min node, and along any root to leaf path nodes alternate between min type and max type.

(ii) The key in any min node is the minimum among all keys in its subtree.

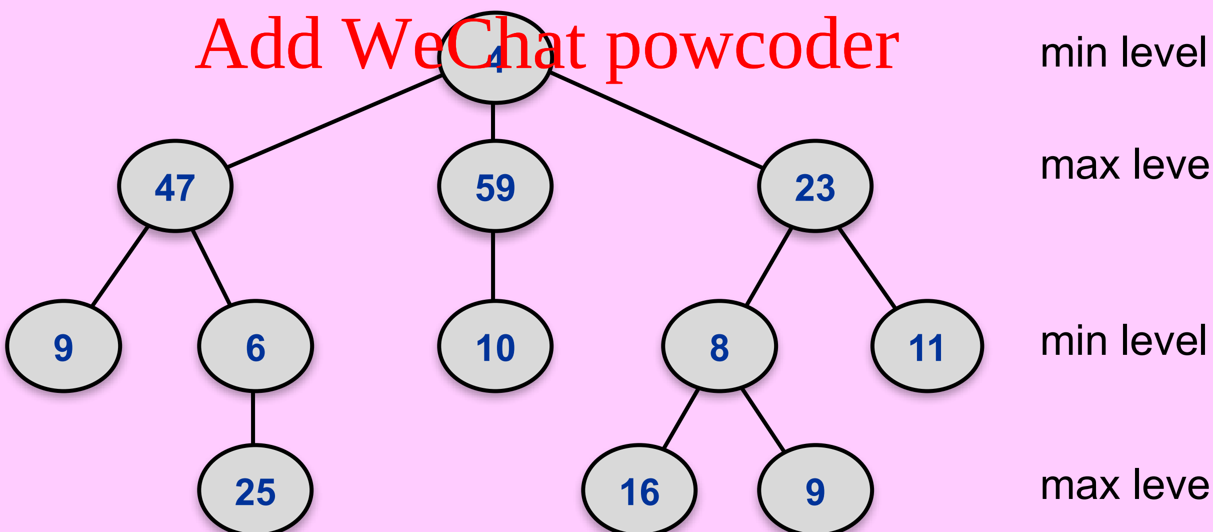
(iii) The key in any max node is the maximum among all keys in its subtree.

(iv) Any additional structural conditions that you may specify.

(a) Suitably complete the specification (i.e., condition (iv)) of the min-max heap.

(b) How do you initialize an empty min-max heap according to your specification?

(c) Design and analyze efficient algorithms for Insert, DeleteMin and DeleteMax on min-max heaps.



17. Augmented Stack and Queue:

Recall that a stack is a Last-In-First-Out list and a queue is a First-In-First-Out list.

(a) Design and analyze a data structure that maintains a stack S under the following operations, each in O(1) **worst-case** time:

Push(x,S): insert item x on top of stack S.

Pop(S): remove and return the top item of stack S (if not empty).

FindMin(S): return the minimum value item on stack S (if not empty).

(b) Design and analyze a data structure that maintains a queue Q under the following operations, each in O(1) **amortized** time:

Enqueue(x,Q): insert item x at rear of queue Q.

Dequeue(Q): remove and return the front item of queue Q (if not empty).

FindMin(Q): return the minimum value item on queue Q (if not empty).

[Hint: See Exercise 4 of our Introductory Slides.]

(c) Improve the running times in the solution to part (b) to O(1) **worst-case** time per operation. [Hint: divide costly computation into small chunks & do a chunk at a time per queue operation.]

END