ECE365: Data Structures and Algorithms II (DSA 2)

Priority Queues and Heaps

Priority Queues

- A **priority queue** is an ADT that supports two operations:
  - *insert*
  - *deleteMin* (or deleteMax, but not both)
- The insert operation inserts a new item into the priority queue
- Every item has a **key**
- The deleteMin (or deleteMax) operation returns the element with the minimum (or maximum) key
- The deleteMin operation also alters the data structure, removing the element
- Sometimes other operations are also implemented

Applications of Priority Queues

- Job scheduling
- Heapsort
- Certain graph algorithms (covered in our next topic)
- Huffman coding (covered later in the course)
- The A* search algorithm (we learn about this in my Artificial Intelligence course)

Simple Implementations of Priority Queues

- Idea 1: Use a simple linked list, inserting new nodes at the start or end
  - insert would take O(1)
  - deleteMin would take O(N)
- Idea 2: Use a linked list, keeping it sorted
  - insert would take O(N)
  - deleteMin would take O(1)
- There are at least as many insertions as deleteMin operations, so the first idea is probably the better of the two
- However, neither idea is good, because one of the two main operations is linear in either case
- Idea 3: Use a balanced binary search tree (e.g., an AVL tree)
  - Both insert and deleteMin would have worst-case logarithmic time
  - This is better than the first two ideas, but it is still overkill (it supports other, unnecessary operations), and it is not as efficient as the next method we will introduce

Binary Heaps

- A **binary heap**, a.k.a. a **heap**, is a very common data structure to implement a priority queue
- Recall that "heap" also has a different meaning in computer science; it can refer to the portion of RAM where dynamic memory is located
- Binary heaps will support both insert and deleteMin in worst-case logarithmic time
- The insert operation will have average-case constant time
- A binary heap will also allow building a priority queue (if all data is provided at once) in worst-case linear time

A Structural Constraint

- A binary heap will be implemented using a **complete binary tree**
- This is a binary tree in which every level of the tree is completely filled, with the possible exception of the bottom level which gets filled from left to right
- It is easy to see that a complete binary tree of height h has between 2^h and 2^{h+1}-1 nodes
- This implies that the height of a complete binary tree with N nodes is the floor of $\log_2 N$, which is clearly $O(\log_2 N)$
- If the maximum possible number of items is known, it is easy to implement a complete binary tree with a regular array
- Otherwise, in C++, a vector can be used, and resized when necessary
- It is common to not store any item in index 0 of the array or vector (we will see that this makes certain operations simpler to implement and a bit more efficient)

Complete Binary Tree Example

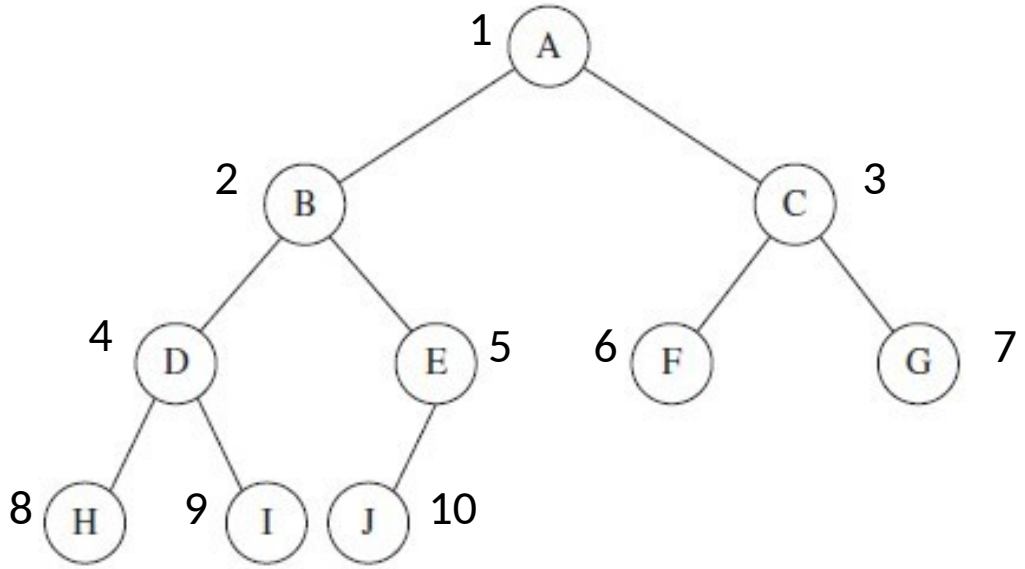


Figure 6.2 A complete binary tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 6.3 Array implementation of complete binary tree

Array-based Complete Binary Trees

- Using array-based complete binary trees, ignoring the zero index will help make operations simpler to implement and a bit more efficient
- The left child of an item stored in array position i will be stored in position 2 * i
- The right child will be stored in position 2 * i + 1
- The index of the parent of a node in slot i is i / 2 (integer division)
- Because of this, bit-shifts and occasional plus-one operations can be used to calculate indexes of parents and children

The Heap Order Property

- As with binary search trees and AVL trees, we will impose not only the structure, but also an ordering property
- For binary heaps, we will call this the **heap order property**
- The heap order property states that the value of the key of each node must be less than or equal to the values of the keys of all children of the node
- This implies that each node has a key with a value smaller than or equal to the keys of all of its descendants
- No node can have a key value smaller than that of the root of the tree
- As with AVL trees, a basic operation may, at first, violate one of the constraints, in this case the heap order property
- We cannot let a heap operation terminate until it is restored

Heap Example (only the left one is valid)

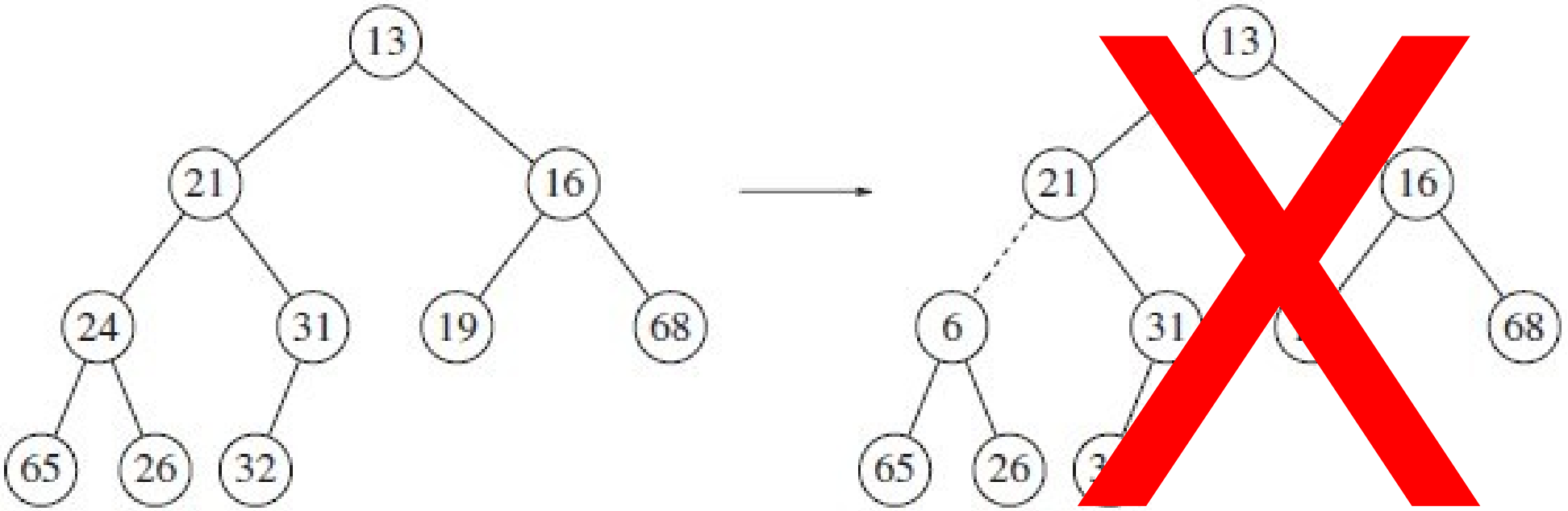


Figure 6.5 Two complete trees (only the left tree is a heap)

Insertion and Percolate Up

- To perform an **insert** operation into a binary heap, there is a specific place that the new item needs to be inserted to maintain a complete binary tree
- However, a simple insertion may violate the heap order property if the value of the key of the new node is smaller than the key of its parent
- If the heap order property is violated, the insertion operation continues by exchanging the newly inserted item with its parent
- This is repeated until the heap order property is restored
- This strategy of fixing the heap is known as a **percolate up**
- To make the insertion more efficient, rather than doing full swaps, elements can move down into a *hole*
- The new element is only copied to the heap after its final location is determined

Insertion Example (insert 14)

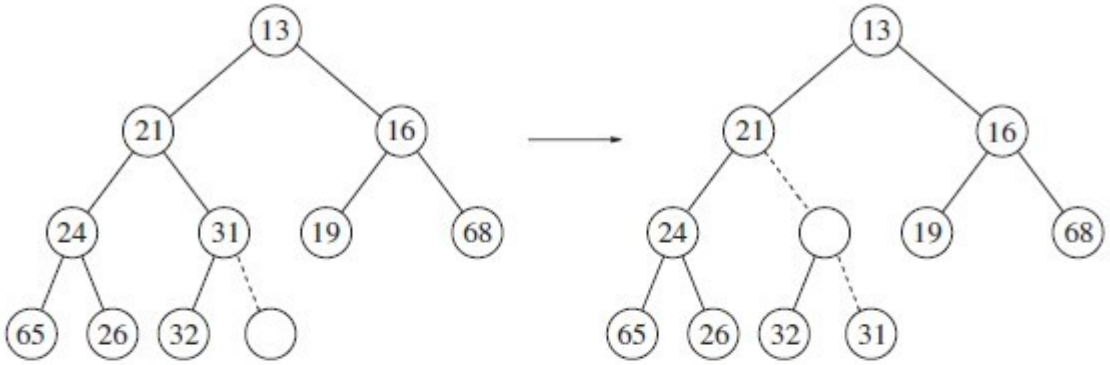


Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

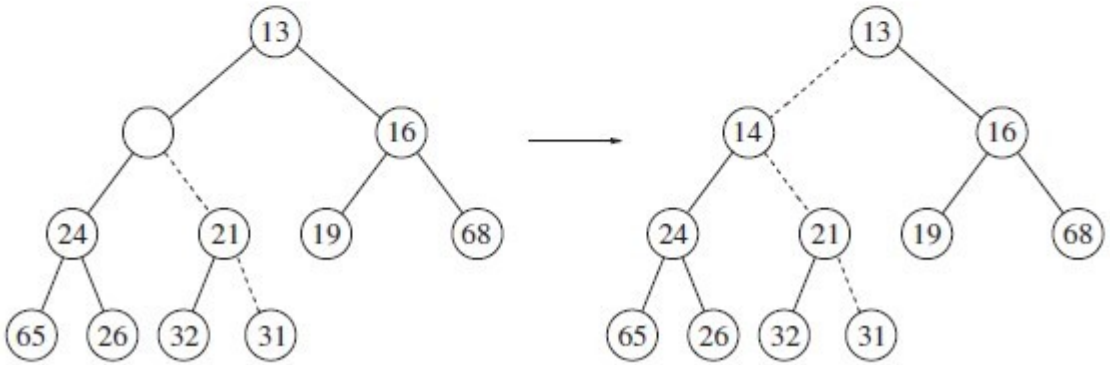


Figure 6.7 The remaining two steps to insert 14 in previous heap

Deletion and Percolate Down

- To perform a **deleteMin** operation, we need to start by finding the minimum element; that's easy – it's at the root!
- Once this is removed, the heap is one item smaller than it was
- Structurally, the last item (i.e., the right-most item at the bottom level) must be removed
- We can start by moving its value to the root; however, that will very likely violate the heap order property
- This element is thus exchanged with the smaller of its two children; this process is repeated until the heap order property is restored
- This strategy is known as **percolate down**
- As with insertion, we can make this more efficient by using the concept of a hole
- Elements are moved up into the hole; the item that started at the last position is only reinserted when its correct location is found

DeleteMin Example (31 is percolating down)

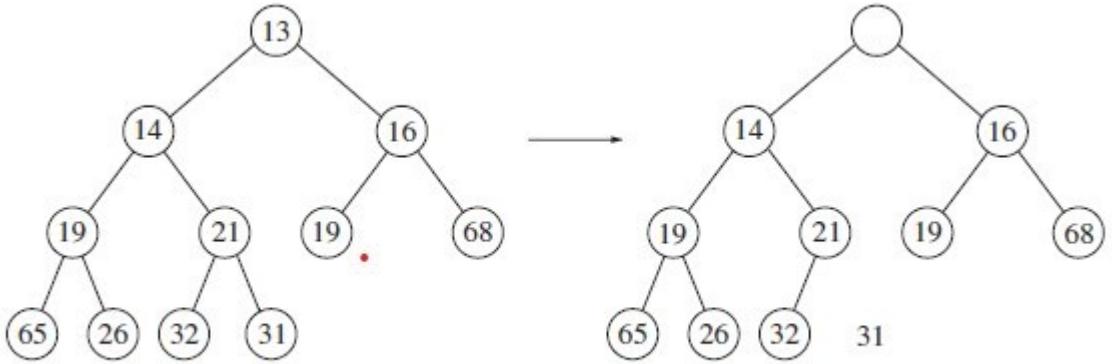


Figure 6.9 Creation of the hole at the root

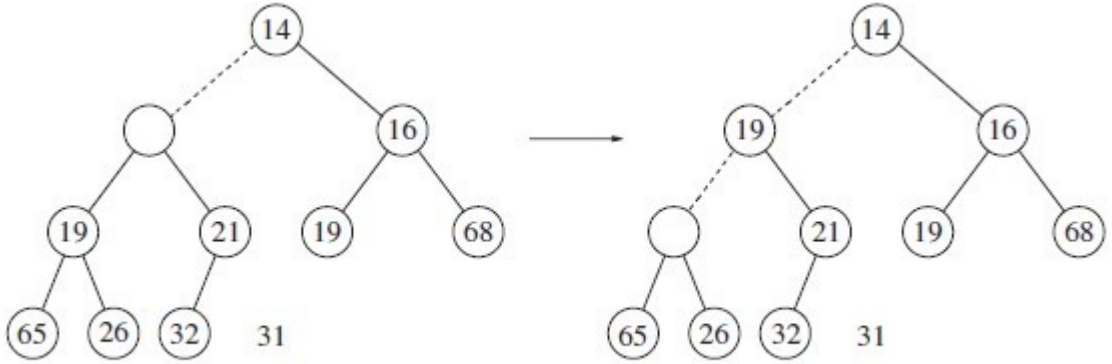


Figure 6.10 Next two steps in `deleteMin`

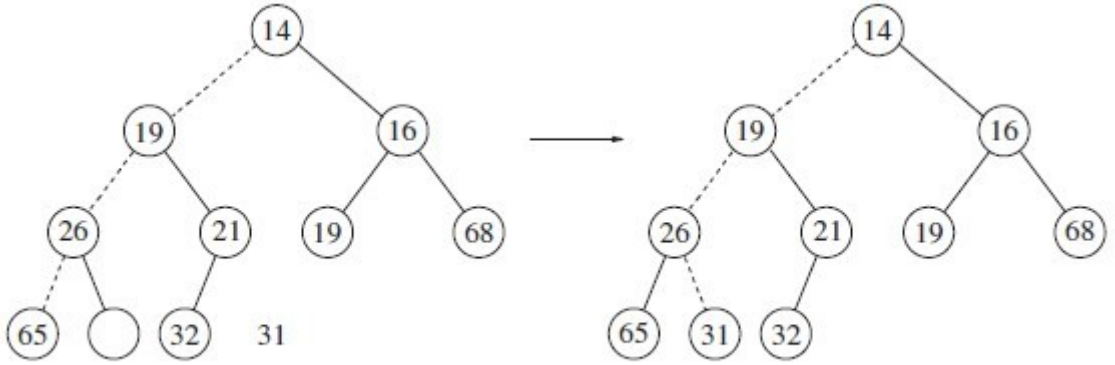


Figure 6.11 Last two steps in `deleteMin`

Be Careful!

- There is a possible implementation error that can occur when percolating down
- You can not assume that every non-leaf node will have two children
- One way to tackle this is to use sentinel values guaranteed to be larger than every valid key
- A simpler way to handle this is to do extra checks, verifying that the locations of the left and right child of the hole are valid locations

Insert and DeleteMin Complexities

- Insert requires no more than $\log_2 N$ comparisons
- DeleteMin requires at most $2 * \log_2 N$ comparisons
- Clearly, these are both worst-case O(log N) operations
- However, insert is often much quicker
- According to the book, the average insertion requires 2.607 comparisons
- This means that the hole is percolated up 1.607 levels on average
- This means that insert takes constant average time

Locating Items in a Binary Heap

- Note that a binary heap contains very little ordering information
- There is no way to find a particular item in the binary heap without a linear scan through the entire heap
- To be able to locate items within a binary heap, you must combine the heap with another data structure such as a **hash table**
- Note that the key used by the hash table to identify an item can be different than the key used by the binary heap to order nodes

Other Heap Operations

- **decreaseKey**
  - Specifies a position (in the array) and a positive amount to decrease the value of the key; this might violate the heap order property, so it requires a percolate up
  - Note that if some sort of id is specified, and not the position, then you would need a hash table or some other extra data structure to locate the item efficiently
- **increaseKey**
  - Analogous to decreaseKey
  - It requires a percolate down
- **remove** (or **delete**)
  - One way to implement this is to first use decreaseKey to change the value to negative infinity (or to the root's key minus one) and then call deleteMin
  - Another approach is to move the final item to the delete node’s location, and then percolate it up or down
- **buildHeap**
  - This takes N items as input and creates a binary heap
  - One way to do this is to simply perform N insert operations; this solution requires worst-case time O(N log N), but average case time O(N), since insertions take constant average time to perform
  - There is another approach that takes worst-case linear time

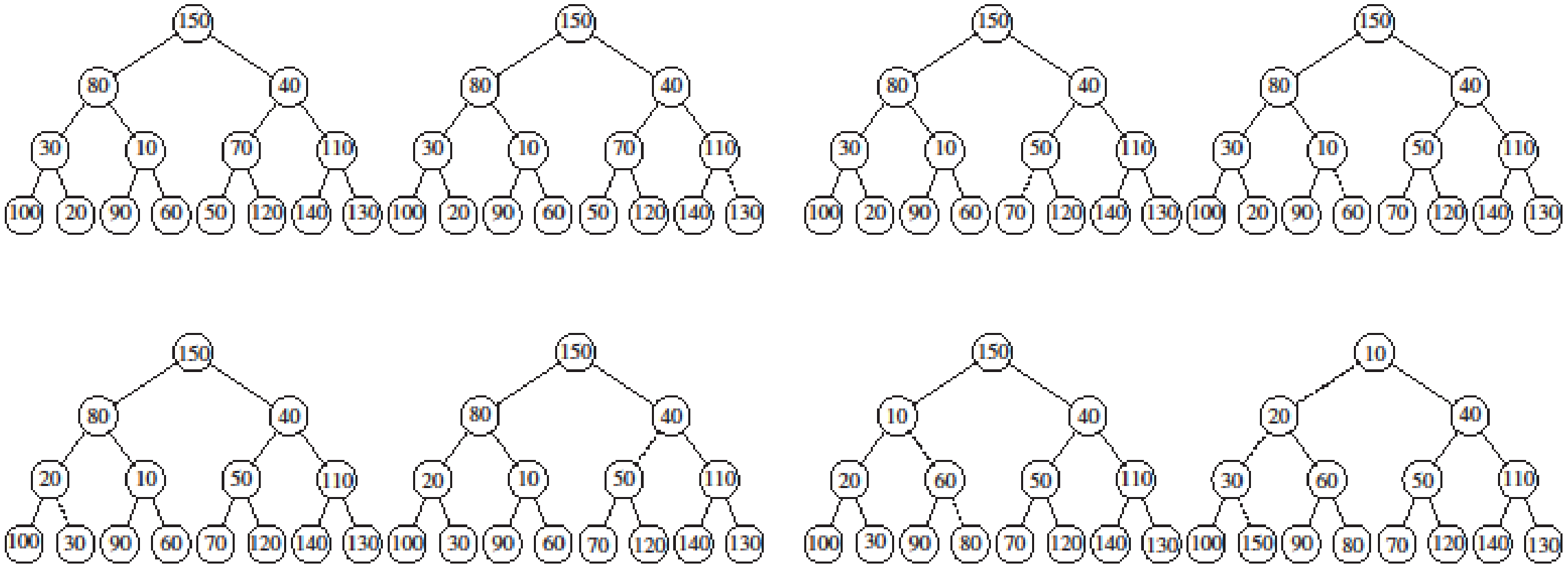
Worst-case Linear BuildHeap

- First, place the items in the array in any order
- Then do the following:

```
Loop i from N / 2 down to 1
  percolateDown(i)
```

- Note that N / 2 (using integer division) is the position of the parent of the final node in the heap

BuildHeap Example



Proving Worst-case Linear BuildHeap

- Consider a *perfect binary tree* (a complete binary tree with a full final level) of height h
- Such a tree has 2^h leaf nodes and $2^{h+1} - 1$ total nodes
- The sum of the heights of all the nodes is: $1*h + 2*(h-1) + 4*(h-2) + \ldots = \sum_{i=0..h} 2^i * (h - i) = 2^{h+1} - 1 - (h + 1)$
- A complete binary tree is not necessarily a perfect binary tree,
- This formula still gives an upper bound on the sum of the heights for any complete binary tree of height h
- Since any complete binary tree has at least 2^h nodes, it is easy to see that the sum of the heights is linear with respect to the number of nodes
- Since each node will be percolated down a number of levels less than or equal to its height, the algorithm is guaranteed to be linear overall

Applications of Priority Queues (Revisited)

- Job scheduling – Trivially simple with a priority queue
- Heapsort
  - First, you can apply a linear buildHeap operation (perhaps using the original memory, if the data is provided in sequential memory)
  - Then, you can perform N worst-case logarithmic deleteMin operations
  - Each removed item can be placed in what was the final heap location
  - This would result in a reverse-sorted sequence; if this is not what is desired, a heap supporting deleteMax (instead of deleteMin) can be used
- Certain graph algorithms (covered in our next topic)
- Huffman coding (covered later in the course)
- The A* search algorithm (we learn about this in my Artificial Intelligence course)

d-Heaps

- One possible extension of heaps is to use *d-heaps* instead of binary heaps
- In a d-heap, each node has d children (so a two-heap is equivalent to a binary heap)
- A d-heap can be much shallower than a binary heap, improving the running time of the insert operation to $O(\log_d N)$
- The deleteMin operation is slower, however, requiring $O(d * \log_d N)$ time; if d is treated as a constant, the asymptotic running time is the same
- Note that if items are stored in an array, finding parents and children can not use bit shifts for multiplication and division unless d is a power of 2
- According to the textbook, there is evidence that 4-heaps may outperform binary heaps in practice

Merge for Priority Queues

- One drawback of binary heaps is that there is no efficient way to **merge** two existing heaps together
- This is important for certain applications (although we will not encounter any in this course)
- Merging two binary heaps with the existing implementation can be done in linear time using the buildHeap operation
- We will now discuss other implementations of priority queues that allow efficient (logarithmic) merges
- All of these implementations involve pointers and dynamic memory allocation
- Therefore, the standard operations will be slower by some constant factor
- It would almost certainly not be possible to do provide a logarithmic merge operation with any array-based priority queue implementation
- Just concatenating two arrays requires linear time

Leftist Heaps

- The first implementation of a priority queue that supports an efficient merge is known as a **leftist heap**
- Like a binary heap, a leftist heap is a binary tree, and it uses the same *heap order property*
- However, it is not a complete binary tree; in fact, it is generally quite unbalanced
- There is an additional property that leftist heaps must follow; this is known as the **leftist heap property**
  - We define the **null path length**, NPL(x), of a node x, to be the length of the shortest path from x to a node without two children
  - If x does not have two children, NPL(x) is 0
  - If x is a null value, we define NPL(x) to be -1
  - The *leftist heap property* states that for every node x in the heap, the null path length of the left child is at least as large as that of the right child

Leftist Heap Example (only the left one is valid)

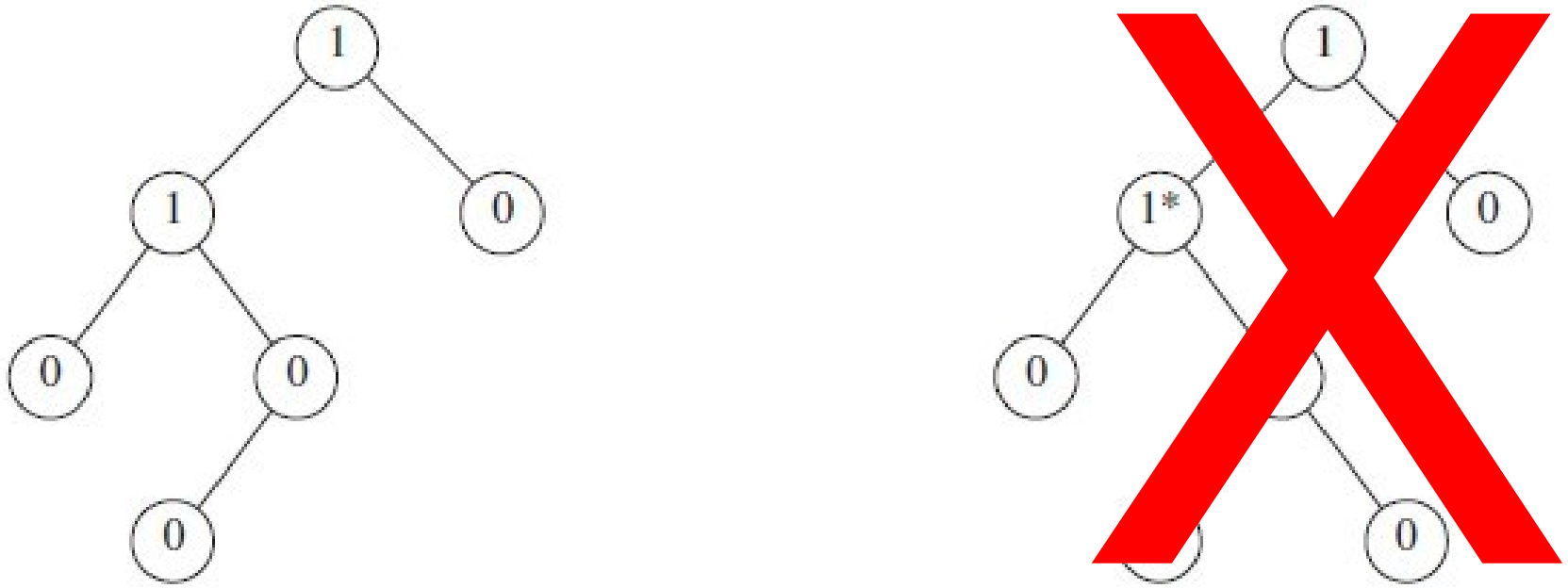


Figure 6.20 Null path lengths for two trees; only the left tree is leftist

Properties of Leftist Heaps

- It should seem intuitive that, due to the leftist heap property, leftist heaps tend to be left heavy; that is, left paths tend to be deeper than right paths
- In fact, a tree consisting of a long path of left nodes is possible
- Because a leftist heap tends to have deep left paths, the right paths tend to be short
- In fact, the right path down a leftist heap is as short as any path from the root to a null
- This means that a leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes; i.e., $N \geq 2^r - 1$
- Therefore, a leftist heap with N nodes has a right path containing at most the floor of $\log_2(N + 1)$ nodes; i.e., $r \leq \log_2(N + 1)$

Leftist Heap Merge Operation (recursive)

- First, we will examine a *recursive solution* for the **merge** of two leftist heaps
  - If one of the heaps is empty, the other heap is returned
  - Otherwise, we compare the roots of the two heaps
  - We then recursively merge the heap with the larger root with the right sub-heap of the heap with the smaller root
  - Assume that the recursive call works in such a way that the merge of these two heaps works correctly without violating either property
  - This step still may violate the leftist heap property at the root of the original heap with the smaller root; this can be fixed with a simple swap of the root's children
- The time to perform the merge is proportional to the sum of the length of the right paths
- Thus, we obtain an O(log N) time bound to merge two heaps

Example: Two Leftist Heaps to Merge

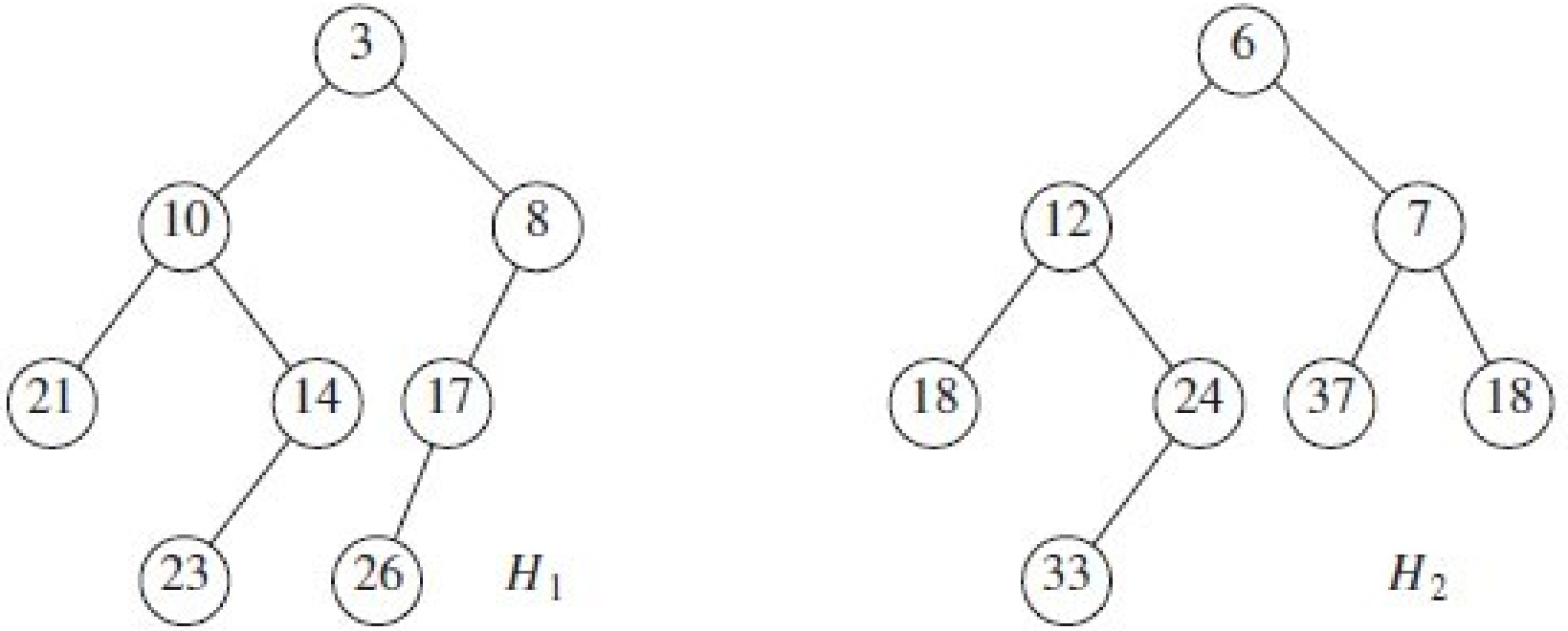


Figure 6.21 Two leftist heaps H_1 and H_2

Example: Merging H_2 and H_1's Right Sub-heap

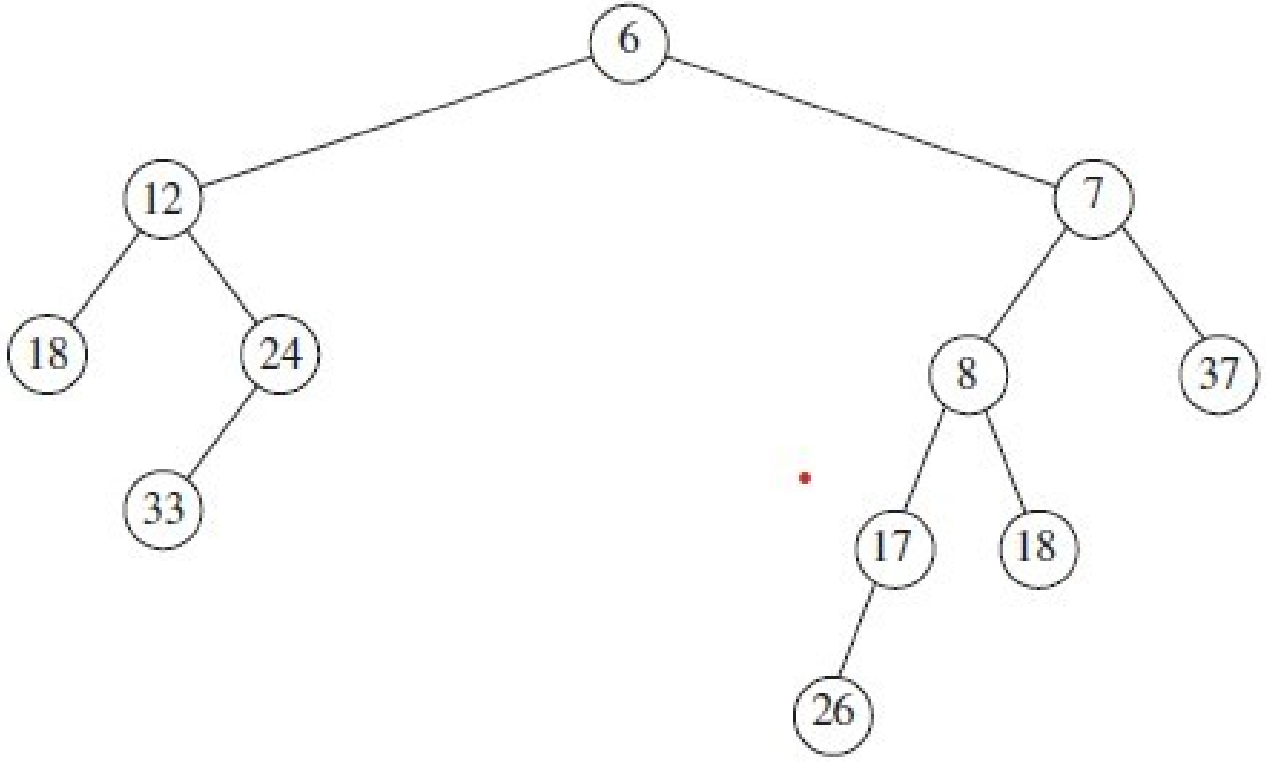


Figure 6.22 Result of merging H_2 with H_1's right subheap

Example: Full Heap After the Recursive Step

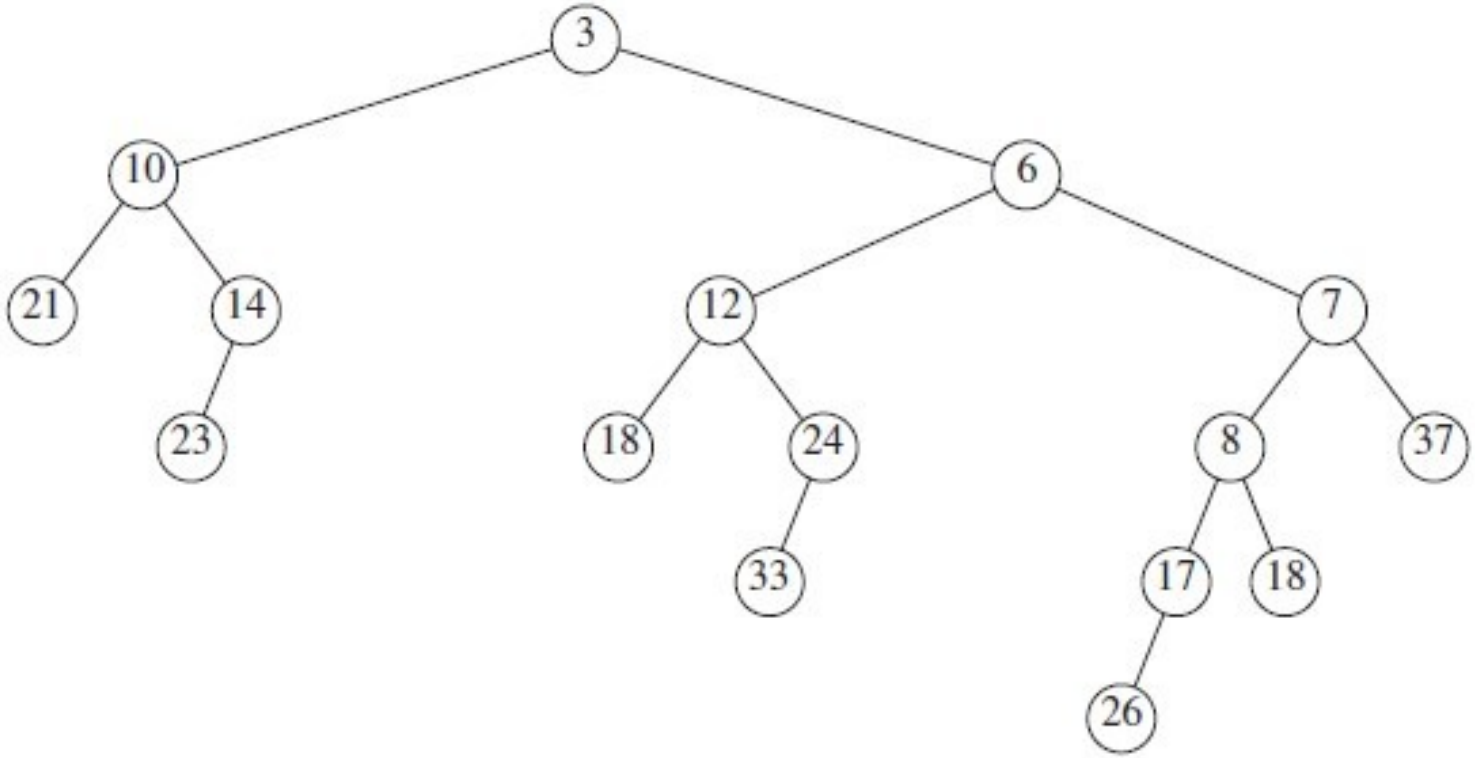


Figure 6.23 Result of attaching leftist heap of previous figure as H_1's right child

Example: Final Result of Leftist Heap Merge

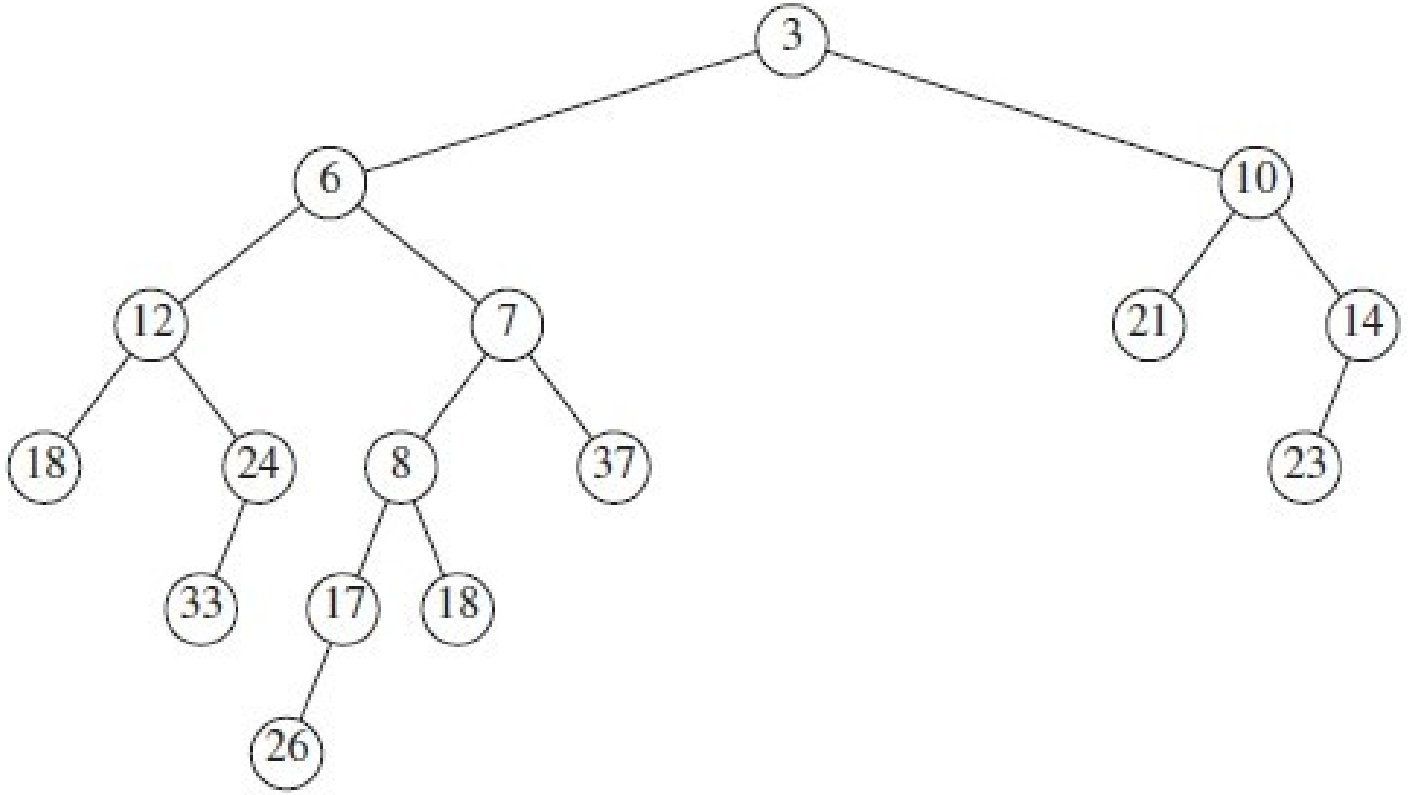


Figure 6.24 Result of swapping children of H_1's root

Leftist Heap Merge Operation (non-recursive)

- Next, we will examine a *non-recursive solution* for the **merge** of two leftist heaps
  - First, merge the two right paths of the two leftist heaps (similar to the merge from mergesort, keeping the nodes' children hanging on)
  - Next, we walk up the new right path, checking for violations of the leftist heap property
  - When we find a violation at a node, we fix it with a swap of the node’s children
- This routine does essentially the same swaps in the same order as the recursive solution

Example: Result of Merging Right Paths

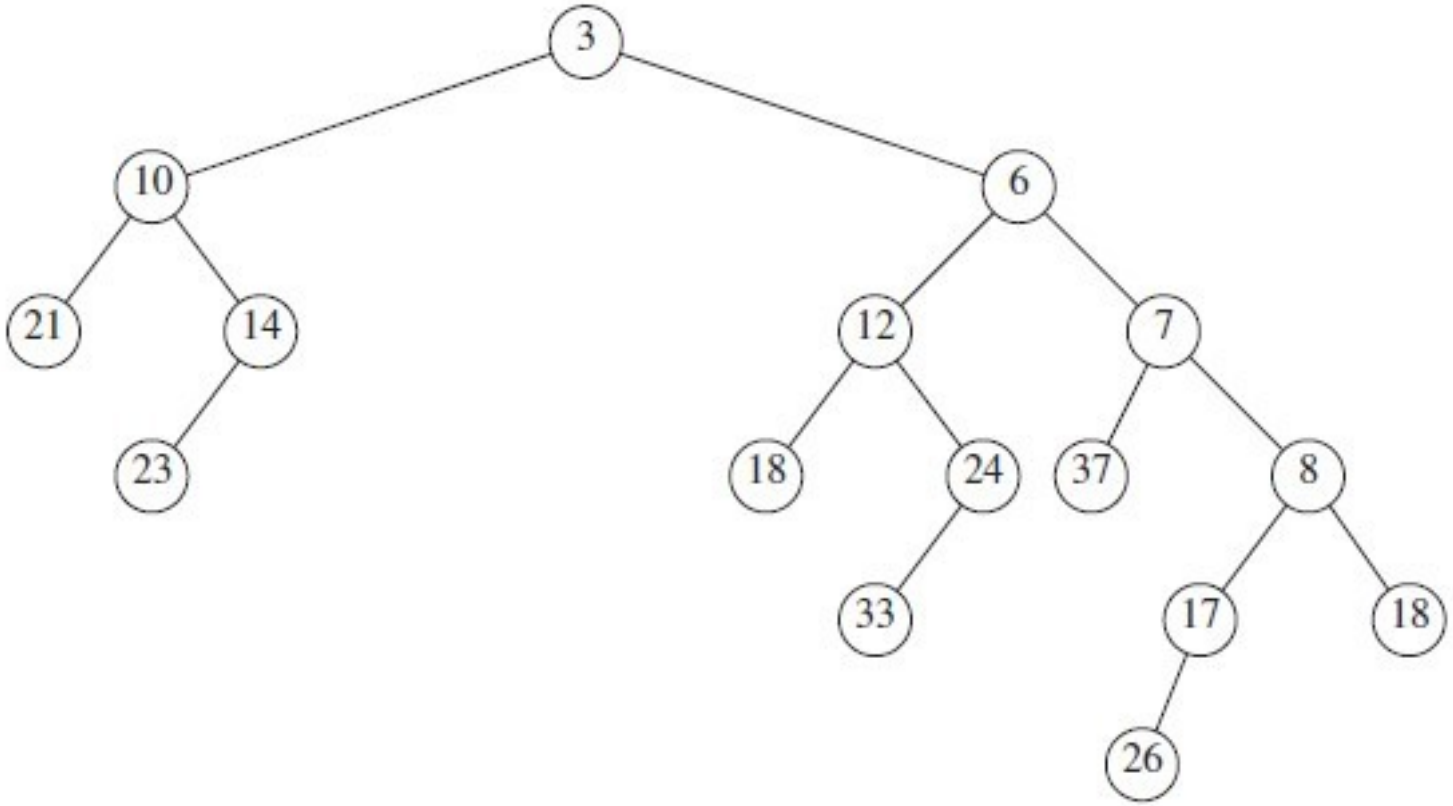


Figure 6.28 Result of merging right paths of H_1 and H_2

Example: Final Result of Merge (again)

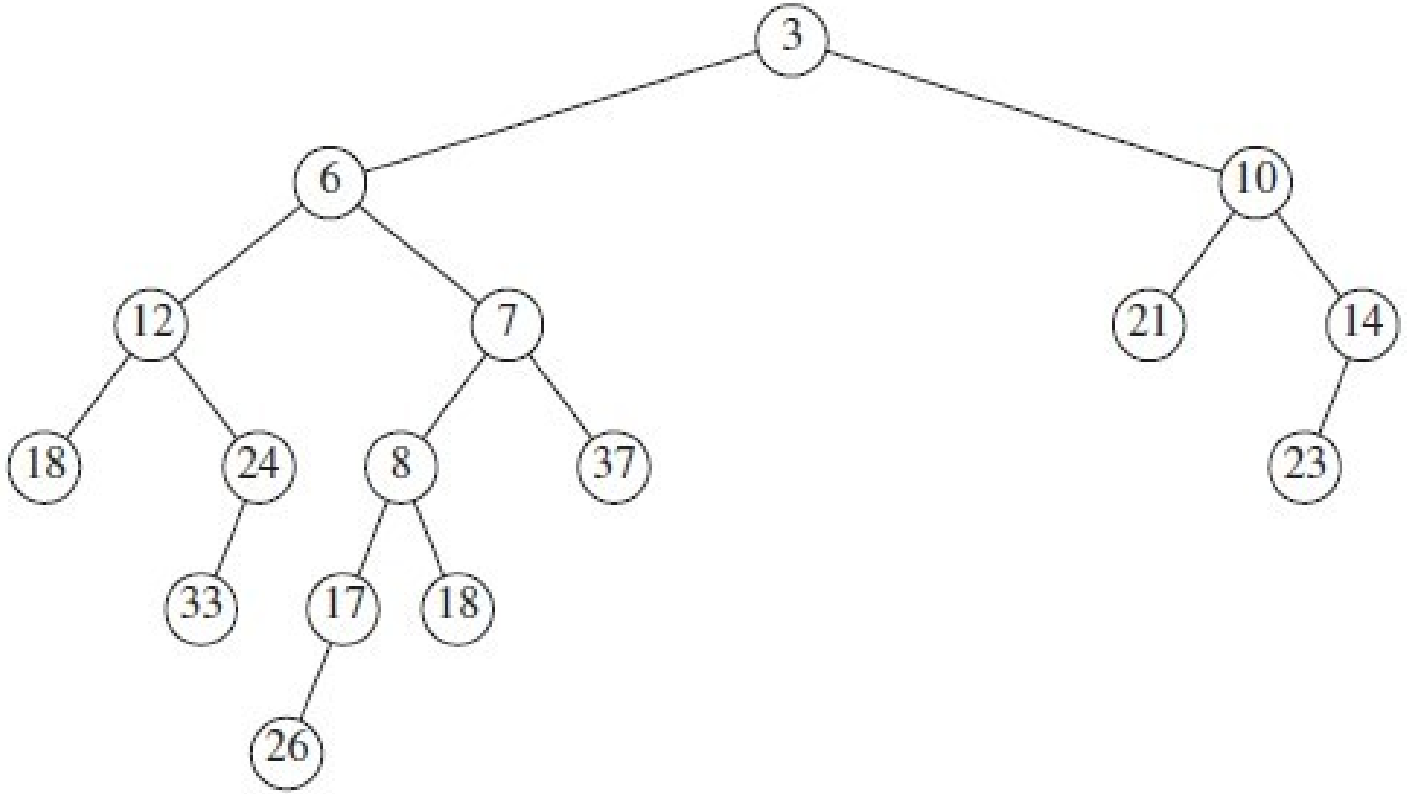


Figure 6.24 Result of swapping children of H_1's root

Insert and DeleteMin for Leftist Heaps

- The **insert** and **deleteMin** operations for leftist heaps can be implemented using the merge operation
- Insert is really just a special case of merge (the new node can be treated as a one-node leftist heap)
- For deleteMin, we remove the root and merge the two sub-heaps

Skew Heaps (briefly)

- A **skew heap** is another implementation of a priority queue that allows an efficient method of merging
- Skew heaps are binary trees with the heap order property but no structural constraint
- No information is maintained concerning null path lengths
- We will not discuss the details of skew heaps, but they have an interesting property
- Skew heaps can become very unbalanced, and the worst-case running time of all operations is O(N)
- However, it can be shown that for any M consecutive operations starting with an empty structure, the total worst-case running time is O(M log N)
- Thus, skew heaps have O(log N) *amortized cost per operation*

Binomial Queues

- A **binomial queues** (a.k.a. **binomial heap**) is another implementation of a priority queue
- Binomial queues also guarantee worst-case O(log N) time per operation for merging, insertion, and deleteMin
- They also provide the further benefit that insertions take constant average time, as with binary heaps
- A binomial queue is not a single heap-ordered tree, but rather a *collection of heap-ordered trees* (general trees, not binary trees, that obey the *heap order property*)
- A collection of trees is also known as a *forest*
- Each of the heap-ordered trees is of a constrained form known as a **binomial tree**
- A binomial tree of height 0 has exactly one node
- A binomial tree, B_k, of height k, is formed by attaching a binomial tree B_{k-1} to the root of another binomial tree, B_{k-1} (the one with the larger root goes lower)
- In a binomial queue, there may be at most one binomial tree of every height

Examples of Binomial Trees

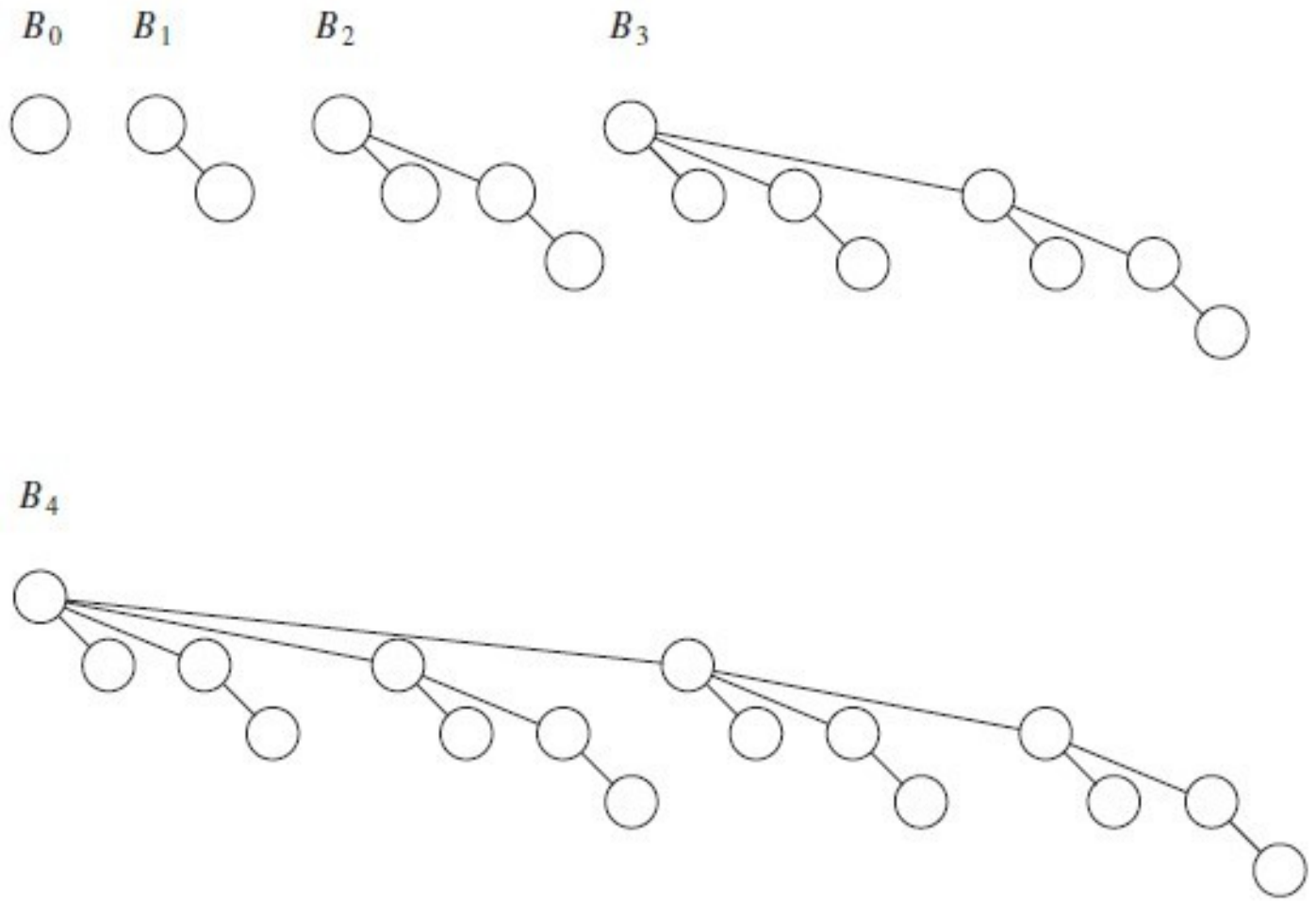


Figure 6.34 Binomial trees B_0, B_1, B_2, B_3, and B_4

Properties of Binomial Trees

- It is simple to see that a binomial tree of height k has exactly 2^k nodes
- Furthermore, in a binomial tree of height k, the number of nodes at depth d is the binomial coefficient ${}_kC_d = k! / (d! * (k - d)!)$
- This is related to the numbers in Pascal's triangle
- We can uniquely represent a priority queue of any size by a collection of binomial trees with different sizes
- This is analogous to representing a number in binary
- As previously mentioned, in a binomial queue, we impose the heap order property on each binomial tree
- We have seen that our textbook draws binomial trees leaning to the right
- Some other sources draw them leaning to the left (which I think may make more sense, given the implementation details, which we will discuss later)

Binomial Queue Merge Operation

- The process of merging two binomial queues is analogous to binary addition
  - The process marches from the smallest binomial trees to the largest, in parallel, in each binomial queue
  - You process the B_0 trees, then the B_1 trees, then the B_2 trees, etc.
  - If there is a single B_k tree, it just moves to the new binomial queue
  - If there are two B_k trees, they combine to form a B_{k+1} tree
  - If there are three B_k trees, two are combined to form a B_{k+1} tree and the third just moves to the new binomial queue
- Combining two binomial trees takes constant time
- There are O(log N) binomial trees, so the merge takes O(log N) time

Example: Two Binomial Queues to Merge

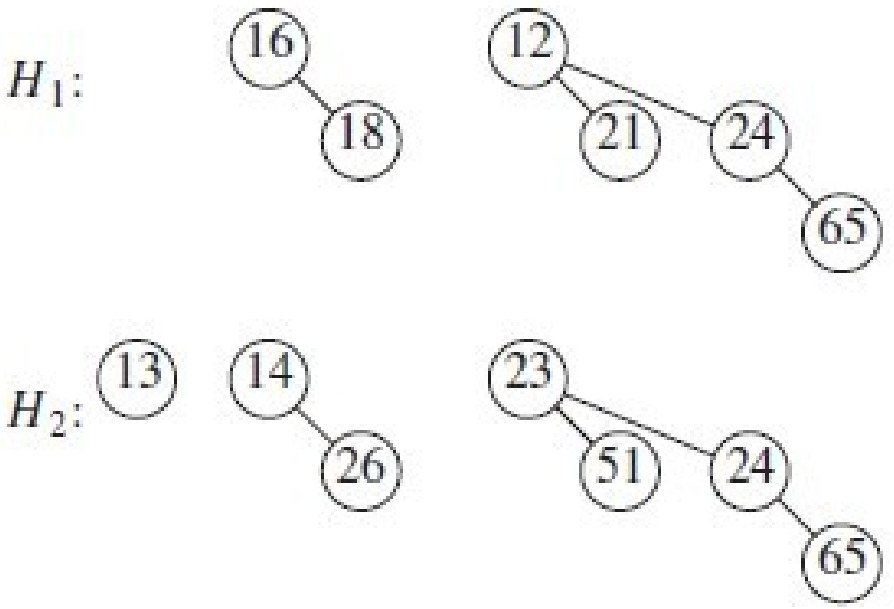


Figure 6.36 Two binomial queues H_1 and H_2

Example: Result of Binomial Queue Merge

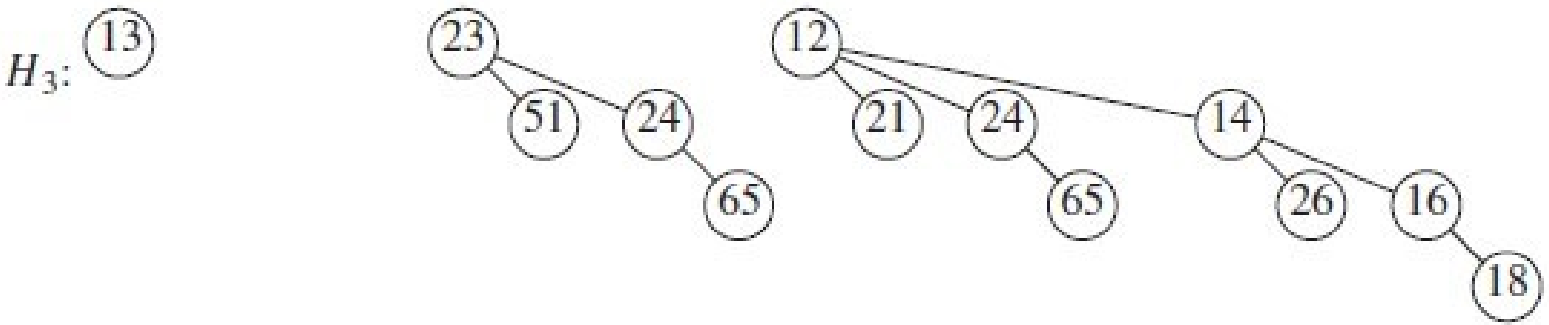


Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

Insertion for Binomial Queues

- As with leftist heaps, **insert** is once again just a special case of merge (the new node can be treated as a one-node binomial queue)
- Therefore, insert takes worst-case logarithmic time
- It turns out that it takes average-case constant time
- Assume that there is a 1/2 probability that a tree of any particular height will be present in an arbitrary binomial queue
- Then, the average insertion will terminate after two steps:

$$(1/2) * 1 + (1/4) * 2 + (1/8) * 3 + (1/16) * 4 + \ldots = 2$$

DeleteMin for Binomial Queues

- As with leftist heaps, **deleteMin** for a binomial queue can be implemented using the merge operation
- The node with the smallest key must be the root of one of the binomial trees (since they all obey the heap order property)
- This can be found in logarithmic time
- When we remove the root from a binomial tree, the result is a binomial queue!
- Therefore, we have two binomial queues to merge:
  - One is the original binomial queue minus the binomial tree that had its root removed
  - The other is the binomial queue formed by removing the root of one binomial tree
- The merge is also logarithmic, so deleteMin is worst-case O(log N)

Implementing Binomial Queues

- Nodes store the leftmost child and right sibling (which is common for general trees)
- Children are arranged in order of decreasing size
- Some sources draw binomial trees as leaning to the left, which I think is more intuitive given this implementation detail
- If each node also stores a pointer to its parent, it is not complicated to implement the decreaseKey operation to run in O(log N) time
- A delete operation can then be implemented in O(log N) time as a decreaseKey followed by a deleteMin

Example of Binomial Queue Implementation

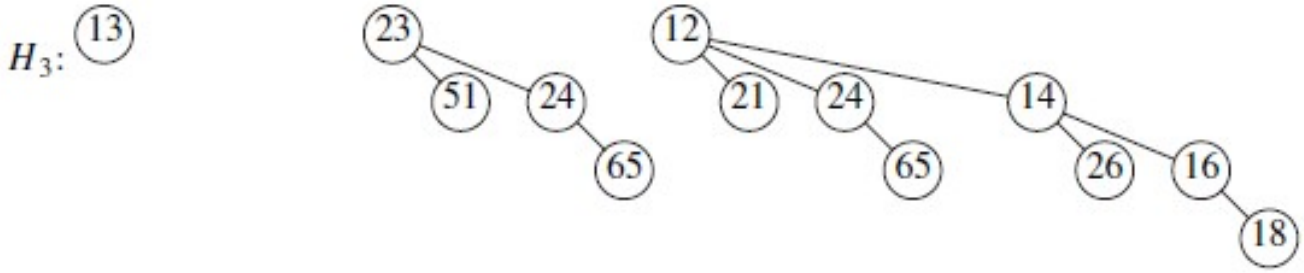


Figure 6.50 Binomial queue H_3 drawn as a forest

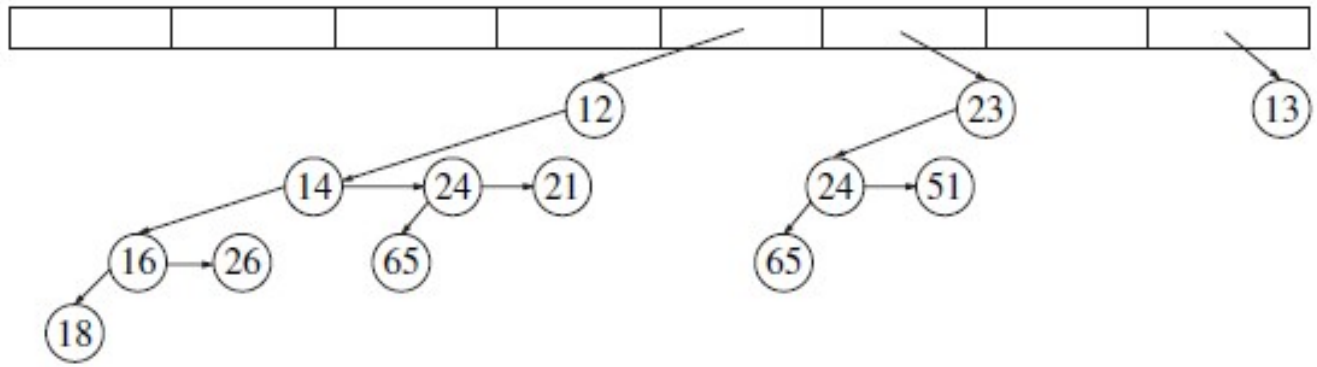


Figure 6.51 Representation of binomial queue H_3

Fibonacci Heaps (briefly)

- A **Fibonacci heap** is another implementation of a priority queue
- We will not discuss the details of Fibonacci heaps, but they have some very nice properties
- As with binomial queues, Fibonacci heaps are collections of trees
- In fact, if the decreaseKey operation and the delete operation are never invoked, each tree will be a binomial tree
- Generally, though, Fibonacci heaps allow a more relaxed structure
- All operations that do not involve deleting an element (e.g., insert, merge, decreaseKey) run in constant amortized time and worst-case logarithmic time
- The other major operations (deleteMin and delete) run in worst-case and average-case logarithmic time
- This can lead to very efficient implementations of certain algorithms