EECS 560
DATA STRUCTURES

MODULE V: PRIORITY QUEUE

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.

-

- If you disagree, please delete the document immediately.

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Data Structures and Algorithm Analysis in C++***, 4th edition, by Mark Allen Weiss.

PRIORITY QUEUE AND ITS APPLICATIONS

- Priority queue is an extension of queue. Unlike the queue ADT which has a first-in-first-out property, the priority queue determines the "first-out" element using a user-defined property that may be different than the enqueue order.

- For example, imagine we are running a server. Rather than simply executing the job that is submitted at the earliest time, we want to execute the job that pays us the most. Here, the property "pays us the most" is defined by us to determine the execution order of the jobs.

PRIORITY QUEUE: FUNDAMENTAL OPERATIONS

- Similar to the queue ADT, the priority queue ADT also has two fundamental operations:
  - enqueue(): insert and element into the queue
  - findHigestPriority() and deleteHighestPriority(): find and delete the element that has the highest priority. Note that these functions are similar to the dequeue() function in the queue ADT. (And in the book they are referred as findMin() and deleteMin())

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using array?
- Can we implement this ADT using linked list?
- Can we implement this ADT using hash table?
- Can we implement this ADT using binary search tree?

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using array?
  - **insert without sorting**: We can directly insert the element to the last position of the array, which will take O(1) time. If the array is not sorted, the we need to go through the entire array to find the element with the highest priority, which will take O(n) time.
  - **insert with sorting**: If the elements in the array are sorted, then we can spent O(log(n)) time to find the correct position for the new element, and then spend O(n) time to move the existing elements to spare a space for the new element. When finding the element with the highest priority, we can simply take the first element using only O(1) time. (Note that if we need to delete the element, we will need an extra O(n) time to move the remaining elements.)

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using linked list?
  - **insert without sorting**: This is the same as if we implement using array. That is, O(1) time for insert and O(n) time for locating the element with the highest priority.
  - **insert with sorting**: We will simply go through the linked list to insert the element to the proper location, which will take O(n) time. (Note that unlike array, linked-list does not allow binary search because we cannot directly access an element by its index in O(1) time.) Accessing the element with the highest priority, including deleting it, will both take O(1) time.

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using hash table?
  - Note that the hash function may not preserve priority. That is, the key with a higher priority does not necessarily mean it will be hashed into a lower (or a higher) position in the hash table. (If yes then it is essentially an array.)
  - In this case, elements in the hash table will not be sorted. Insertion will thus simply take O(1) time; and the search/deletion will take O(n) time.

PRIORITY QUEUE: ANALYSIS

- Can we implement this ADT using binary search tree?
  - Yes, we will be table to do search/insertion/deletion in O(log(n)) time. In this case, inserting an element and finding the element with the highest priority will both take O(log(n)) time.

PRIORITY QUEUE: MOTIVATION

- Can we do better?
- For array, linked list, and hash table, all of them requires an O(n) time for either an insertion or finding the element with the highest priority.
- For binary search tree, both insertion and search will take O(log(n)) time. However, we observe that many operations are wasted on maintaining the BST properties. While it allows the search of any element in O(log(n)) time, we only need to search for the one with the highest priority. So, can we make the BST simpler to allow faster time for searching the element with highest priority?

PRIORITY QUEUE: MOTIVATION

- Our target:
  - O(log(n)) time for insertion
  - O(1) time for finding the element with the highest priority
  - O(log(n)) time for deletion

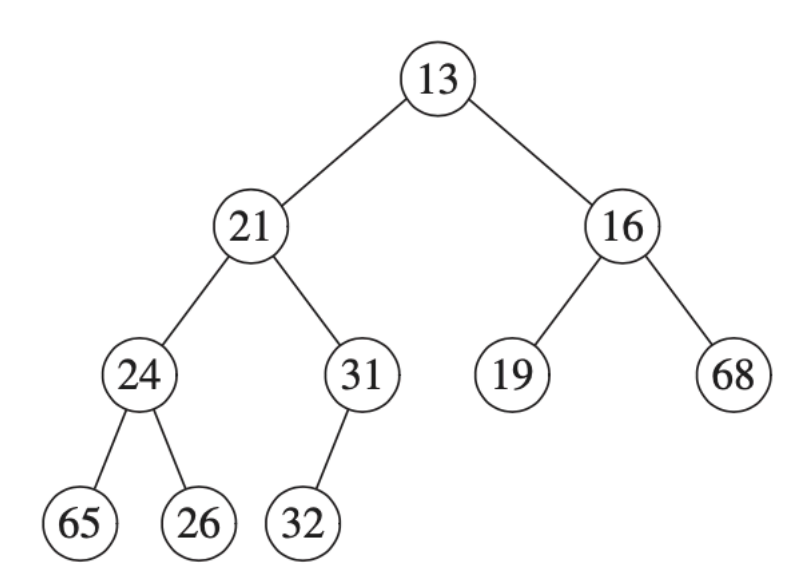
- Expected advantage over BST
  - O(1) time access vs. O(log(n)) time access of the element with the highest priority
  - will use pure array implementation (no pointer and no scattered data blocks in the hard disk), hence more computational and space efficient

PRIORITY QUEUE: HEAP IMPLEMENTATION

- In many cases, "priority queue" and "heap" are used interchangeably. I personally prefer to use "priority queue" to describe the property and behavior of the ADT, and use "heap" in the context of implementation.

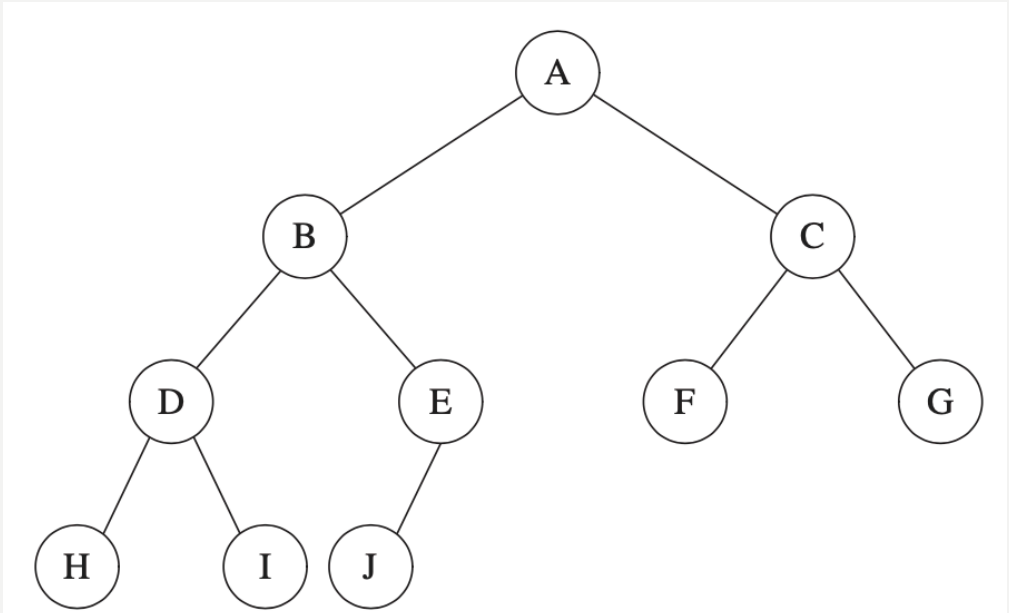
HEAP PROPERTY

- A heap looks like a binary search tree, however it has different properties:
  - a heap is full (elements attached from left to right, layer by layer) while a BST is only balanced (referred as **structure property**)
  - in heap, the parent is smaller than its children (and we don't care which child Is larger); in BST, the parent is larger than its left child and smaller than its right child (referred as **heap-order property**)



HEAP IMPLEMENTATION

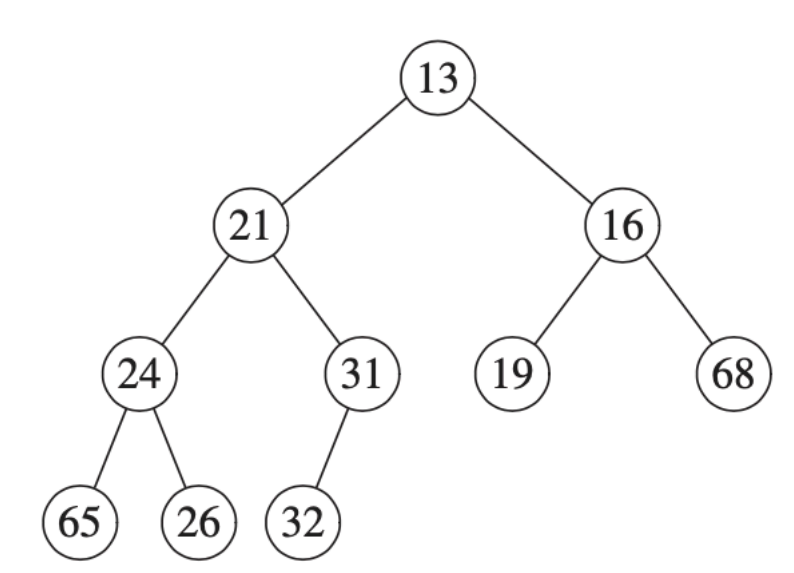
- Because the heap is full, we can use an array to store it (and get rid of the space-consuming pointers).
- For any element, we can always derive its index in the array from its position in the hypothetical binary tree



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

HEAP OPERATION: SEARCH

- Finding the element with the highest priority (consider the numbers in the examples as the "rank", so the element with the highest priority correspond to the smallest number)
  - This is trivial with the heap property, which states that any parent is smaller than its children. In this case, the root will be the element with the highest priority (note that we do not delete it; will discuss deletion later)



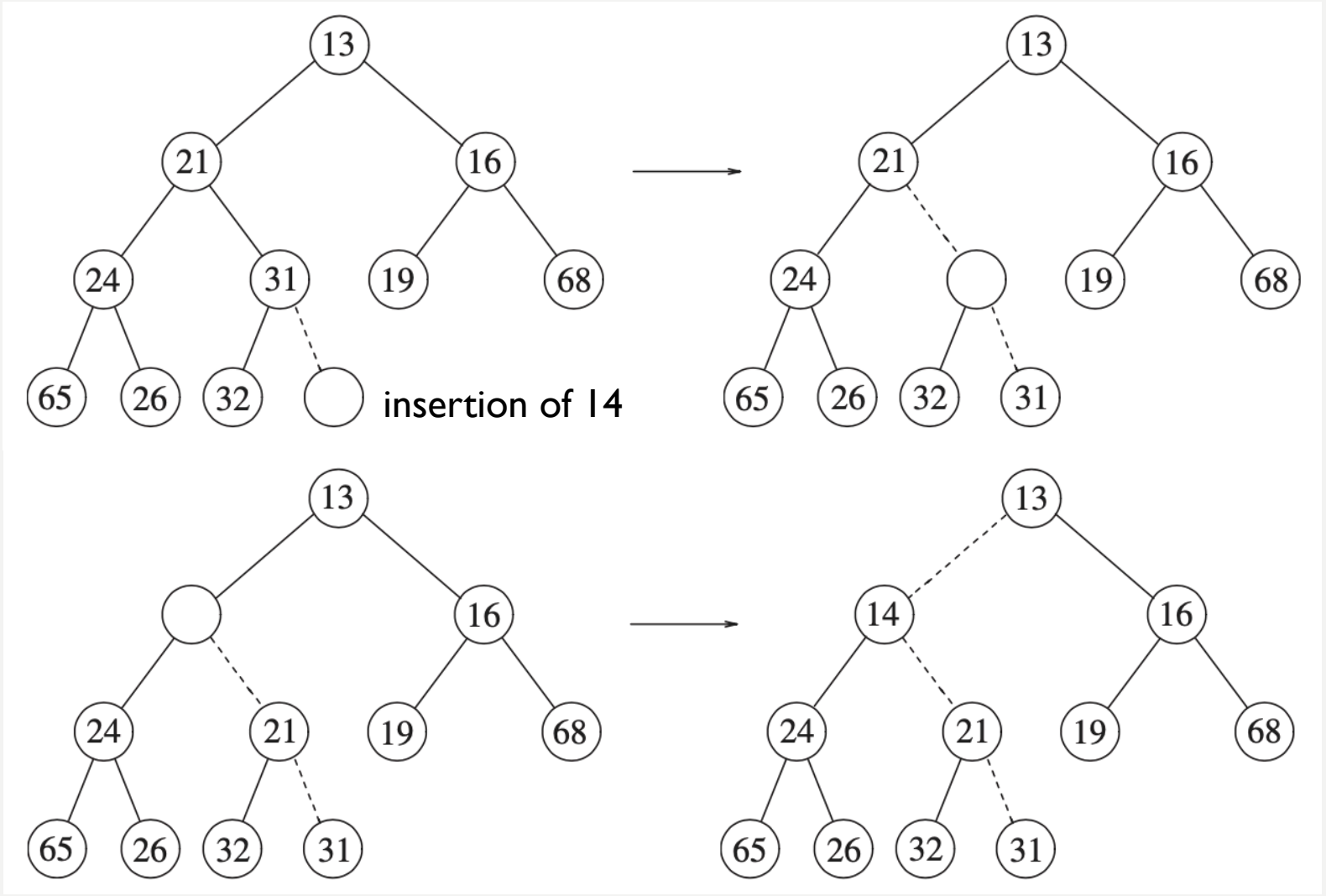
HEAP OPERATION: INSERT

- insert(): Note that the element we insert may have an arbitrary priority.
- Given the structure property of heap (that is, the binary tree is full), the element must be placed on the right-most position of the lowest layer of the tree.
- However, placing the new element on this position may violate the heap property (for example, it may be smaller than its parent). Therefore, we need to adjust the tree if necessary. The process is called "**percolation**".

HEAP OPERATION: INSERT

- We know that if the newly-inserted element is larger that its parent, then we are done (because we respect the heap property).
- Otherwise, we need to "**percolate up**" the newly-inserted element recursively until we find a location that respects the heap property.
- The upward percolation is simple, we just recursively compare the new element with its parent. If the new element is smaller, then we swap it with its parent; otherwise we stop.

HEAP OPERATION: INSERT



HEAP OPERATION: INSERT

```
1       /**
2        * Insert item x, allowing duplicates.
3        */
4       void insert( const Comparable & x )
5       {
6           if( currentSize == array.size( ) - 1 )
7               array.resize( array.size( ) * 2 );
8
9               // Percolate up
10          int hole = ++currentSize;
11          Comparable copy = x;
12
13          array[ 0 ] = std::move( copy );
14          for( ; x < array[ hole / 2 ]; hole /= 2 )      // compare with its parent
15              array[ hole ] = std::move( array[ hole / 2 ] );
16          array[ hole ] = std::move( array[ 0 ] );
17      }
```

HEAP OPERATION: INSERT

- The upward percolation resembles climbing from the leaf to the root. Because the heap is full, the height of the tree is O(log(n)). In this case, the time complexity for upward percolation is also O(log(n)).

HEAP OPERATION: DELETION

- delete(): In the context of priority queue, we are only interested in deleting the element with the highest priority (not any one).

- We know that the element with the highest priority resides in the root. Our task is thus to remove the root.

HEAP OPERATION: DELETION

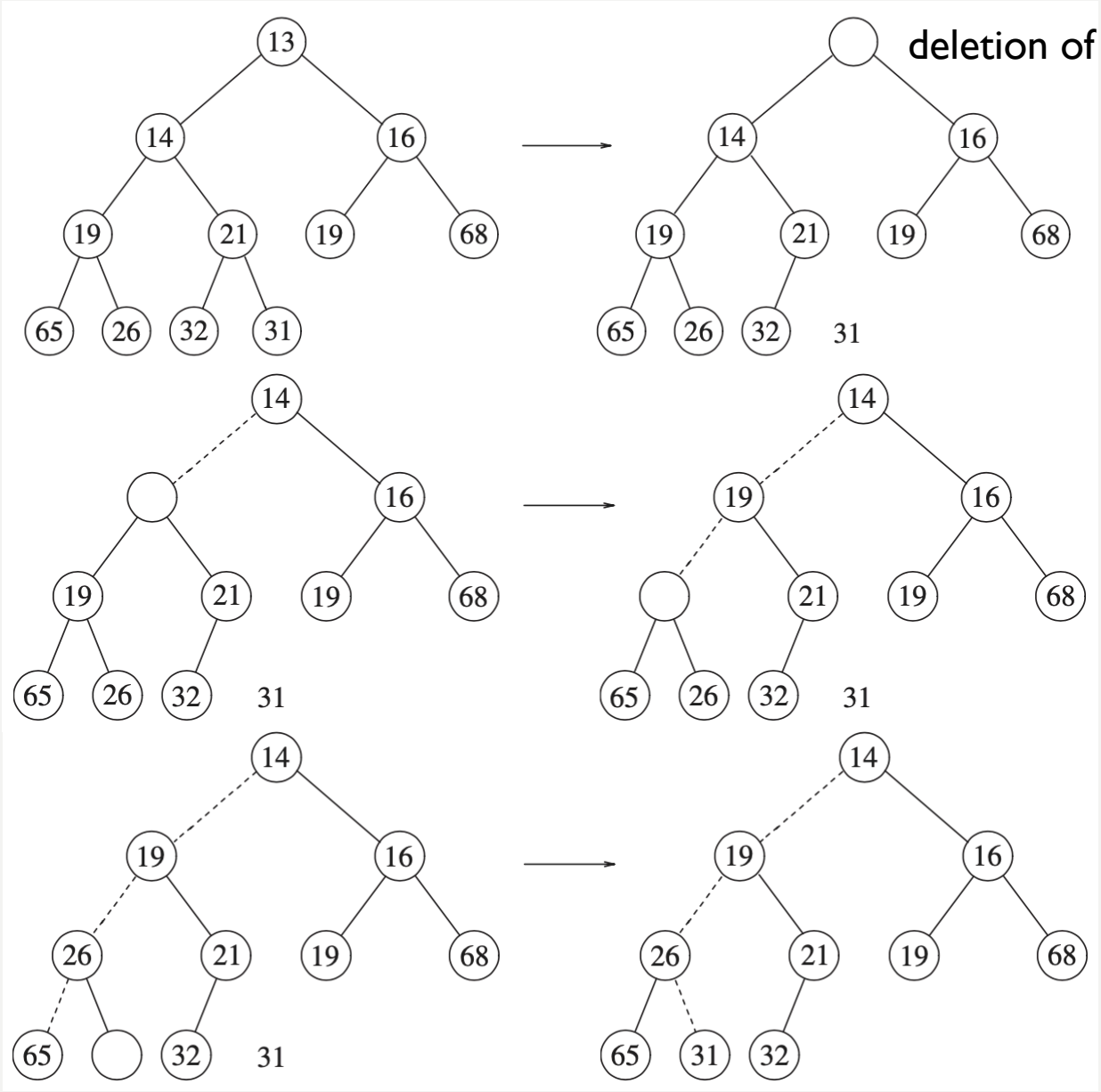
- Once we remove the root, the heap is one element less and we should shrink the array size by 1.
- It expels the last element, and we put the last element in the root as its temporary home.
- However, doing so may violate the heap property (e.g., the last element could be larger than one of its children). In this case, we need to "**percolate down**" the last element to restore the heap property.

HEAP OPERATION: DELETION

- Among the three elements (the element we need to percolate down and its two children), the new parent should be the smallest element among the three.

- In this case, we compare the new element with its smaller child, if the smaller child is smaller than the new element, we swap them. Otherwise we stop.

HEAP OPERATION: DELETION



HEAP OPERATION: DELETION

```
28      /**
29       * Internal method to percolate down in the heap.
30       * hole is the index at which the percolate begins.
31       */
32      void percolateDown( int hole )
33      {
34          int child;
35          Comparable tmp = std::move( array[ hole ] );
36
37          for( ; hole * 2 <= currentSize; hole = child )
38          {
39              child = hole * 2;
40              if( child != currentSize && array[ child + 1 ] < array[ child ] )   // compares with the smaller child
41                  ++child;
42              if( array[ child ] < tmp )
43                  array[ hole ] = std::move( array[ child ] );
44              else
45                  break;
46          }
47          array[ hole ] = std::move( tmp );
48      }
```

HEAP OPERATION: OTHERS

- Other supported operations (with minor modification to the existing percolation algorithm) include:
  - **decreaseKey(p, d):** decrease the element at position p (in the array) by an amount of d. This can be done by percolating the decreased element up.
  - **increaseKey(p, d):** increase the element at position p by an amount of d. This can be done by percolating the decreased element down.
  - **remove(p):** remove the element at position p. This can be done by calling decreaseKey(p, -INF) and deleteMin() (which resembles dequeue()).

HEAP CONSTRUCTION

- Naïve way: we can insert the elements into the heap one-by-one. We know that each heap insertion can be done in O(log(n)) time. In this case, inserting n elements would take O(nlog(n)) time.

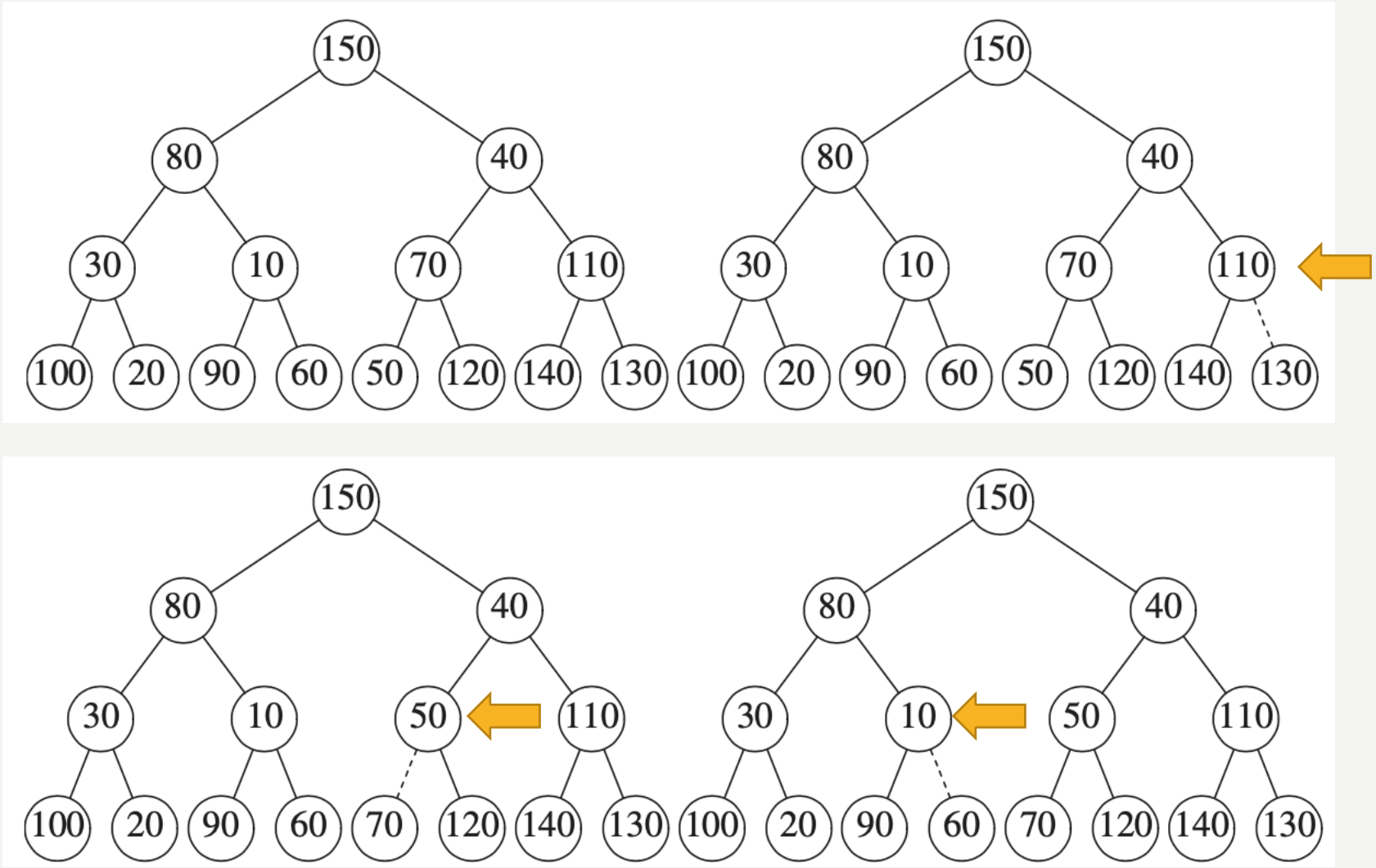
- Can we do better?
  - Given the n elements (in arbitrary order), we wish to construct a heap for them in O(n) time

LINEAR TIME HEAP CONSTRUCTION

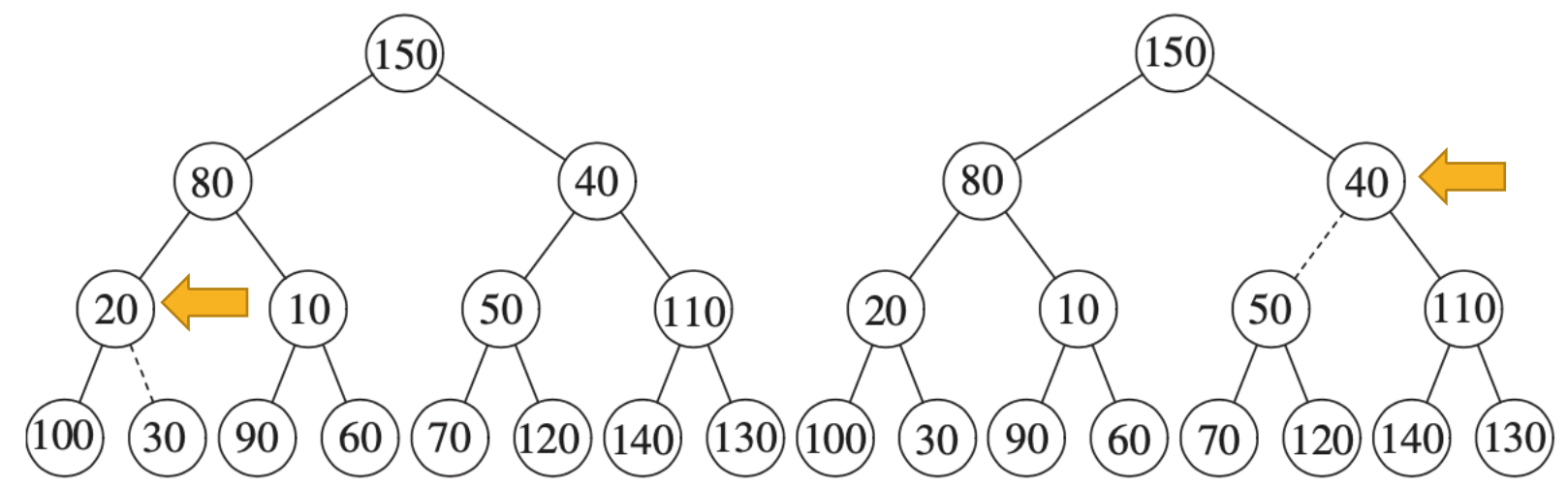
- The algorithm:
  - we copy all elements into the array (using O(n) time); it indicates that we randomly place all elements into the binary tree
  - then, from the last element who is not a leaf to the root, we perform downward percolation (more detailed time analysis will follow)
  - done!

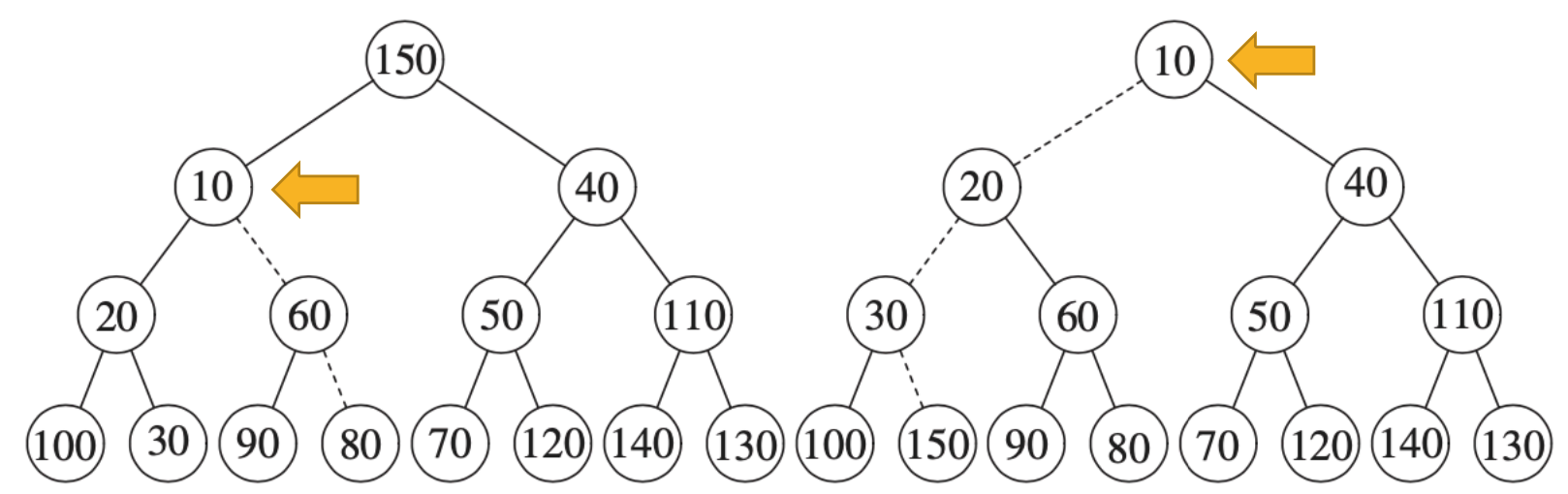
- This is correct because we have essentially percolate down all elements, and finally the resulted heap must have the heap property.

LINEAR TIME HEAP CONSTRUCTION



LINEAR TIME HEAP CONSTRUCTION





LINEAR TIME HEAP CONSTRUCTION

```
 9      /**
10       * Establish heap order property from an arbitrary
11       * arrangement of items. Runs in linear time.
12       */
13      void buildHeap( )
14      {
15          for( int i = currentSize / 2; i > 0; --i )
16              percolateDown( i );
17      }
```

LINEAR TIME HEAP CONSTRUCTION

- Because heap is full, we expect that approximately half of the nodes are leaf; and we don't need to do anything to them.
- Recall that we perform downward percolation from the lower layers to the upper layers. While lower layers contain more nodes, they will also incur fewer steps for each downward percolation.
- Intuitively, we perform "quick" downward percolation for most of the nodes, and "slower" downward percolation for only few of the nodes. Overall our algorithm will be very efficient.
- Now we will go through the formal proof.

LINEAR TIME HEAP CONSTRUCTION (PROOF)

- Theorem: A completely full binary tree has $2^{h+1} - 1$ nodes, the sum of their heights is $2^{h+1} - 1 - (h + 1)$
  - Note that the total heights will serve as an upper bound for the downward percolation
  - That is, the total number of downward percolation will not exceed the total heights
  - If this is proved, then we can prove the O(n) running time, because on average the number of percolation associated with each node is O(1) (divide the total heights by the total number of nodes)

LINEAR TIME HEAP CONSTRUCTION (PROOF)

- Proof:
  - Note that we have 1 root, and its height is 2^h (the 0th layer).
  - In the next level (the 1st layer), we have 2 nodes, and their heights are $h - 1$.
  - In the next level (the 2nd layer), we have $2^2 = 4$ nodes, and their heights are $h - 2$
  - ……
  - In the ith layer, we have 2^i nodes, and the height of each node is $2^{h-1} - i$

LINEAR TIME HEAP CONSTRUCTION (HEAP)

- Proof cont.
  - the total height:$S = \sum_{i=0}^{h} 2^i(h-i) = h + 2(h-1) + 4(h-2) + 8(h-3) + \cdots + 2^h(1)$
  - multiply it by 2: $2S = 2\sum_{i=0}^{h} 2^i(h-i) = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \cdots + 2 * 2^h(1)$
  - we notice that many of the terms will actually cancel out
  - if we subtract the first equation from the second, we will get: $S = 2 + 4 + 8 + \cdots + 2^h - h = 1 + 2 + 4 + 8 + \cdots + 2^h - (h+1) = 2^{h+1} - 1 - (h+1)$

MERGING HEAPS: MOTIVATION

- One limitation of the existing heap implementation is that it is may take a lot of time to merge two heaps.
- Essentially we will construct a new heap using both of their elements, which requires a linear time as discussed above.
- Can we do better? Imagine that each heap respects the structural and heap properties, and we could develop a smarter way to merge them without disrupting individual heaps.
  - Several implementations that allow O(log(n)) merge

MERGING HEAPS: MOTIVATION

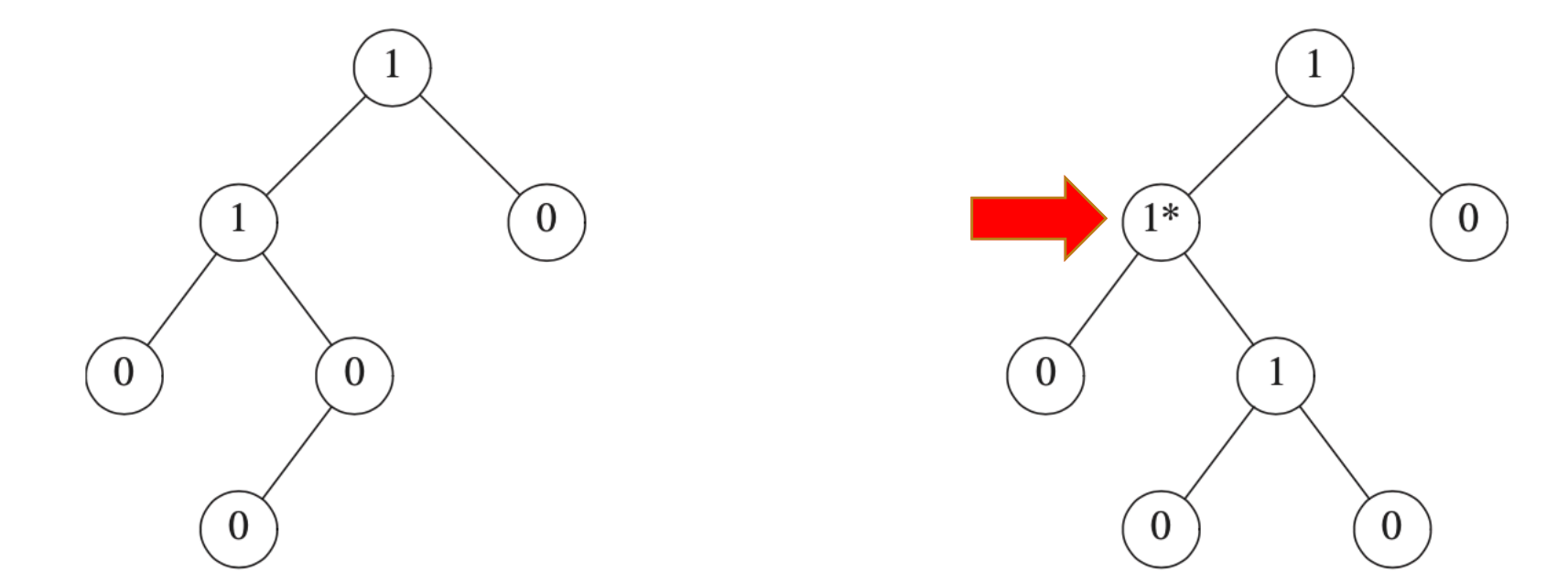
- Two enhanced heap data structures that support faster merge
  - leftist heap
  - binomial heap

LEFTIST HEAP

- The leftist heap has the same heap property, that is, the parent is smaller than its two children.
- However, the leftist heap has a different topology; it is not necessarily full.
- The leftist heap property:
  - define the **null path length** of a node as the shortest path length between itself and one of its descendent leaf
  - for each node in the leftist heap, **the null path length of its left child must be at least as large as its right child**

LEFTIST HEAP

- The left one is a leftist heap, while the right one is not.
- The null path length of the right child of the highlighted node is larger than the null path length of its left child.



LEFTIST HEAP: FUNDAMENTAL OPERATIONS

- Note that the leftist heap is still a heap. That is, it still needs to perform the fundamental top(), enqueue(), dequeue() operations efficiently (more specifically, top() in O(1) time, enqueue() in O(log(n)) time and dequeue() in O(log(n)) time).

- The top() operation can be performed within O(1) time through returning the root element.

- The enqueue() and dequeue() operations are performed through **merge**():
  - enqueue(): **merge** the existing leftist heap with the leftist heap that contains the sole element to insert
  - dequeue(): remove the root, and **merge** the remaining two subtrees

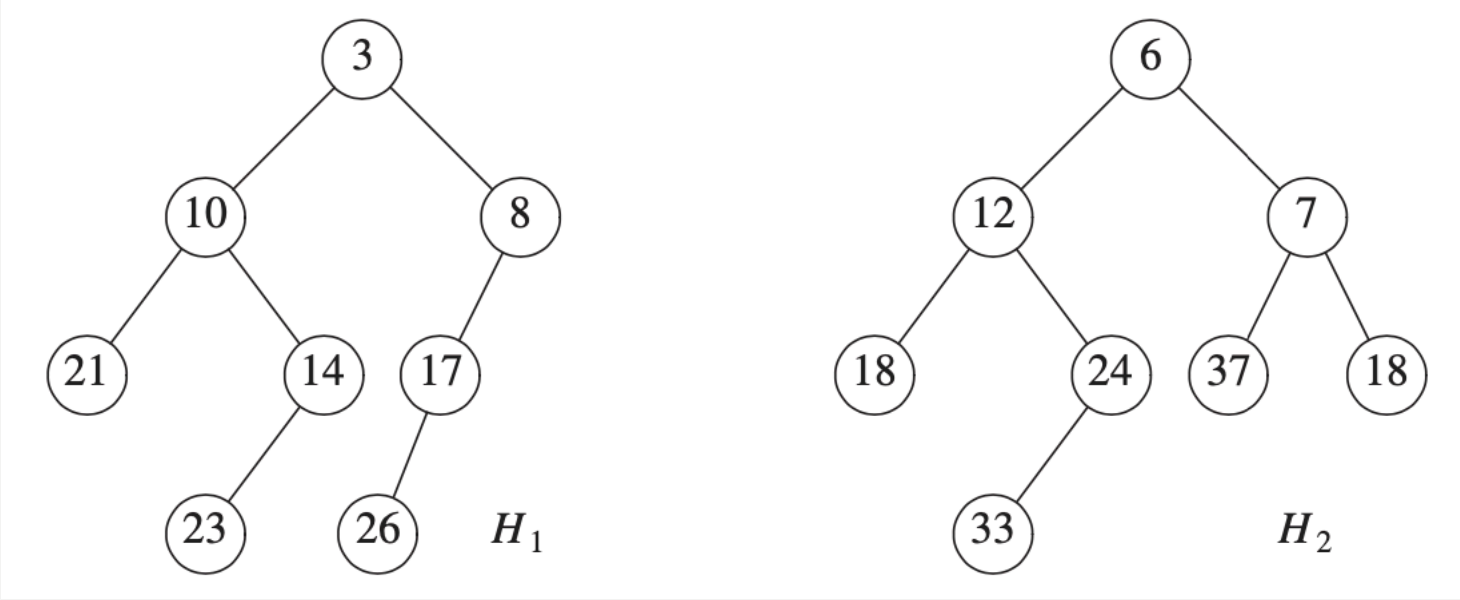
LEFTIST HEAP: MERGE

- It is fair to say, that the fundamental operations of the leftist heap is **merge**.

- The merge will take two **recursive** steps:
  - merge the heap with a larger root with the right subtree of the heap with a smaller root
  - if the null path length of the right subtree is smaller than the null path length of the left subtree, swap the two subtrees

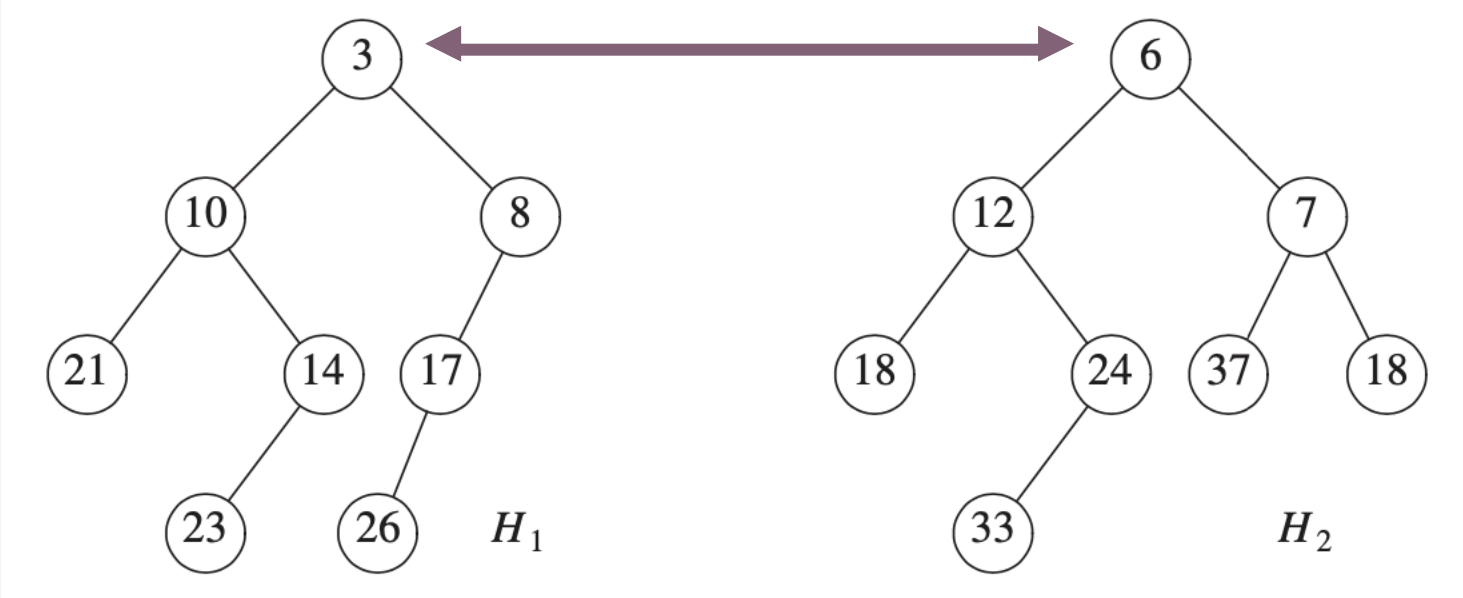
LEFTIST HEAP: MERGE

- imagine we are merging the following two subtrees



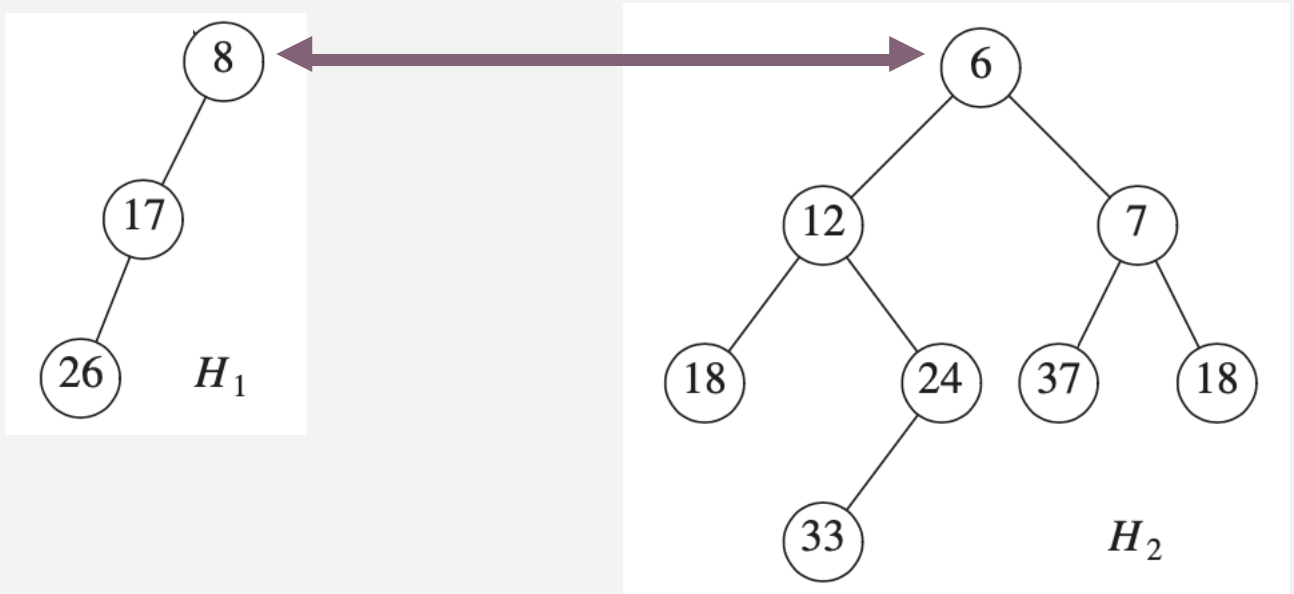
LEFTIST HEAP: MERGE

We compare the roots, the right heap has a larger root.
So, we merge it with the right subtree of the left heap.



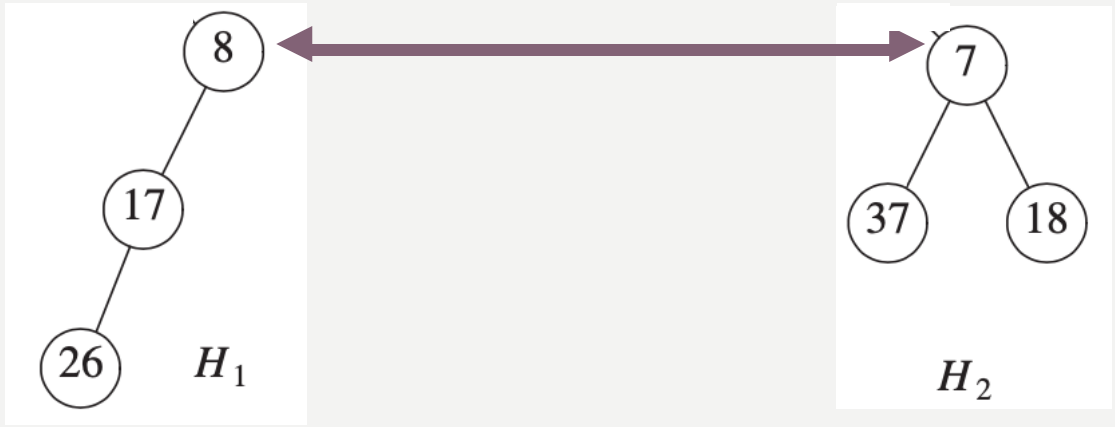
LEFTIST HEAP: MERGE

We compare the roots, the left heap has a larger root.
So, we merge it with the right subtree of the left heap.



LEFTIST HEAP: MERGE

We compare the roots, the left heap has a larger root.
So, we merge it with the right subtree of the left heap.

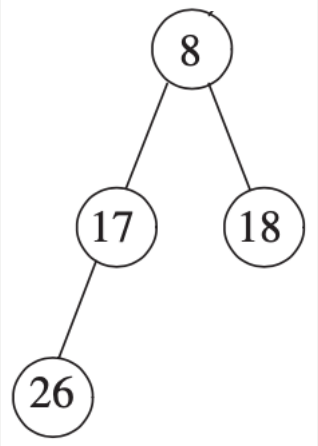


LEFTIST HEAP: MERGE

We compare the roots, the left heap has a larger root.
So, we merge it with the right subtree of the left heap.

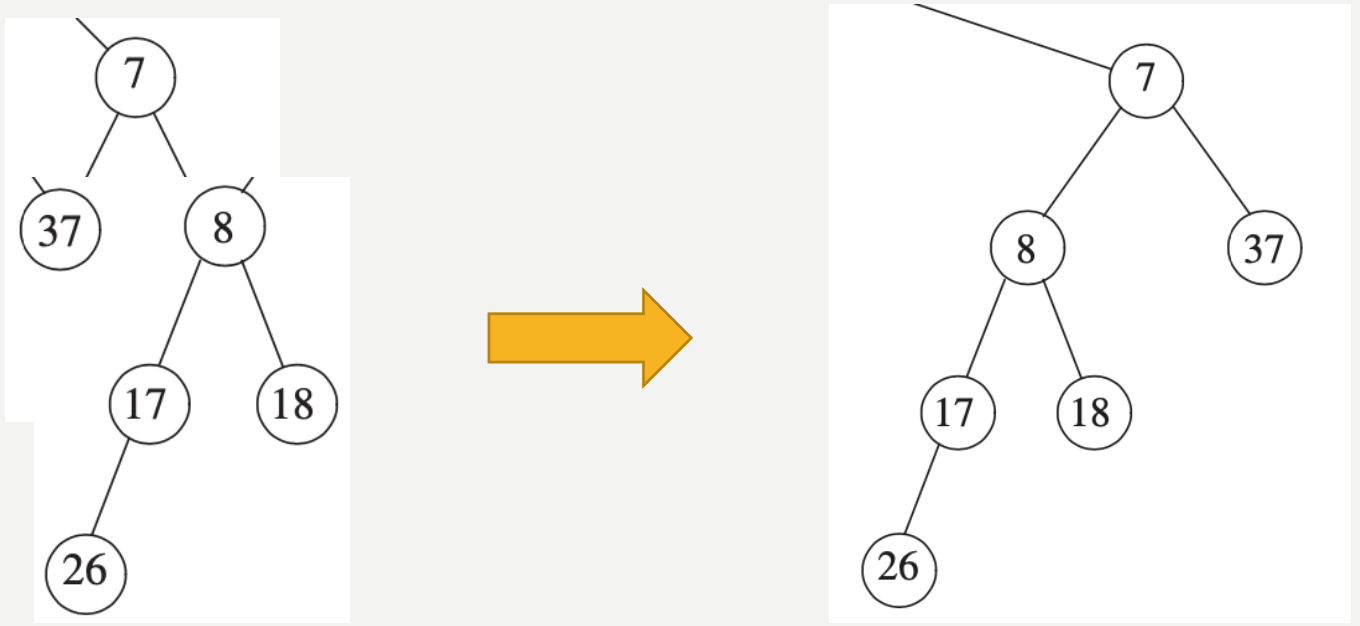
LEFTIST HEAP: MERGE

Now we can merge them!!!



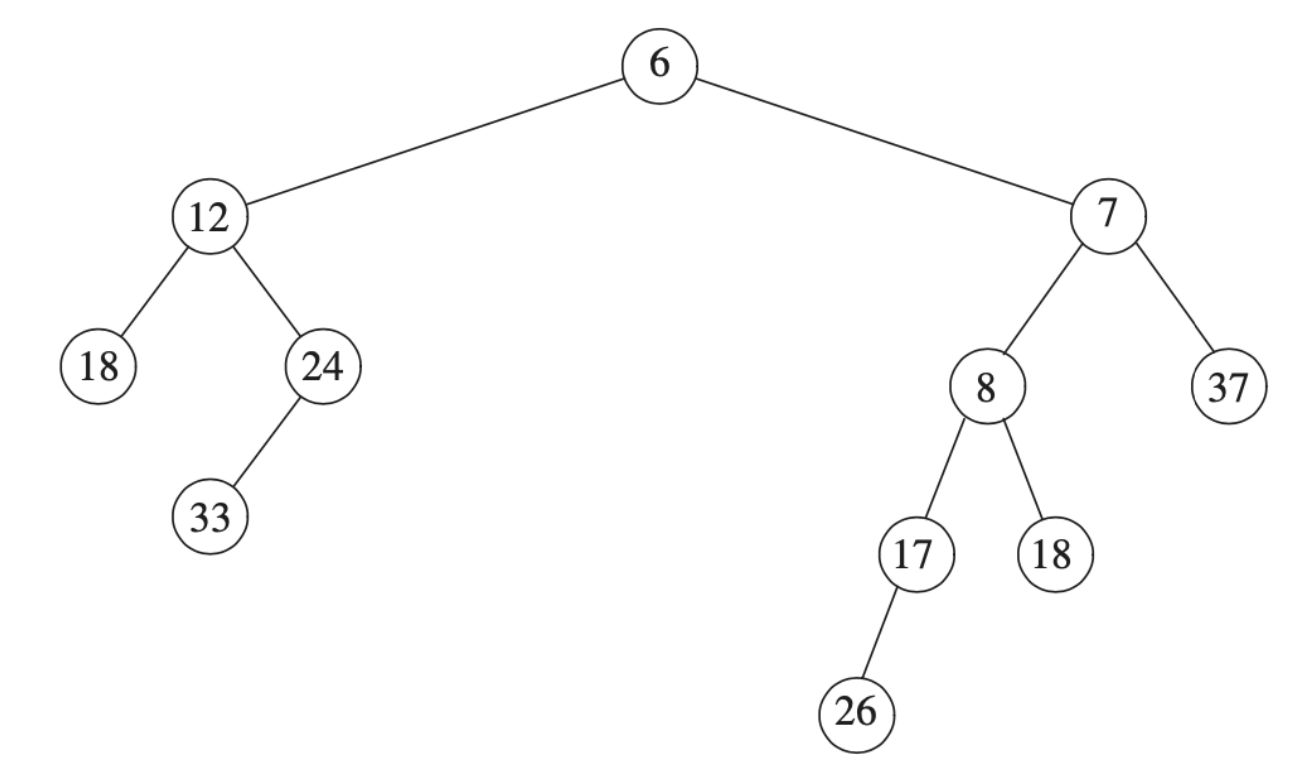
We have to check if the left subtree has a larger null path length.
Checked, no swapping is needed.

LEFTIST HEAP: MERGE



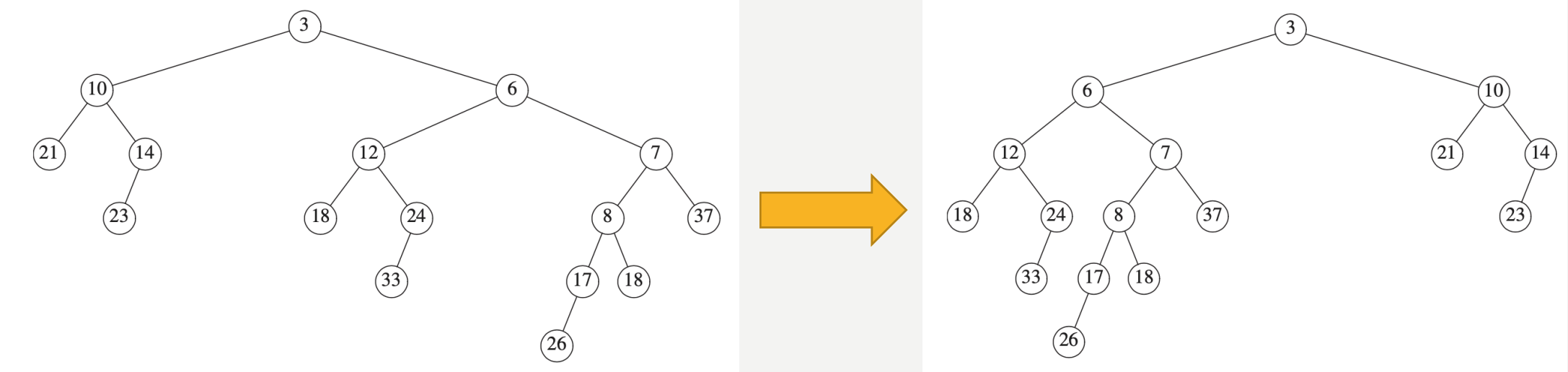
We have to check if the left subtree has a larger null path length.
No, so we swap them.

LEFTIST HEAP: MERGE



We have to check if the left subtree has a larger null path length.
Checked, no swapping is needed.

LEFTIST HEAP: MERGE



We have to check if the left subtree has a larger null path length.
No, swap them.
And we are done!

LEFTIST HEAP: MERGE

```
1       /**
2        * Internal method to merge two roots.
3        * Assumes trees are not empty, and h1's root contains smallest item.
4        */
5       LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 )
6       {
7           if( h1->left == nullptr )   // Single node
8               h1->left = h2;          // Other fields in h1 already accurate
9           else
10          {
11              h1->right = merge( h1->right, h2 );
12              if( h1->left->npl < h1->right->npl )
13                  swapChildren( h1 );
14              h1->npl = h1->right->npl + 1;
15          }
16          return h1;
17      }
```

// key assumption

// boundary case: we attach h2 as the left child to ensure the leftist property

// recursive merge in deeper levels

// swap if necessary
// update null path length

LEFTIST HEAP: MERGE

- Time complexity? We expect O(log(n)).
- If we can prove the time complexity, we can indirectly show that enqueue() and dequeue() will both take O(log(n)) time. This is because each of them will only incur a single merge operation.
- Recall the merge process, all operations (comparison and swapping) are performed on the right subtree.
- Therefore, we would like to show the time complexity through bounding the right subtree height.

LEFTIST HEAP: MERGE

- Theorem: A leftist tree with r nodes on the right path must contain at least $2^r - 1$ nodes.

- Proof (informal): Intuitively, according to the leftist tree property (the null path length of the left child is larger than the null path length of the right child), the tree should be skew to the right (i.e., higher left subtrees and shorter right subtrees). In this case, if the right path has r nodes, then all other paths must have at least r nodes. The tress is then with a height of r; and such a tree clearly has at least $2^r - 1$ nodes.

LEFTIST HEAP: MERGE

- Proof (formal): The proof is by induction. If $r = 1$, there must be at least one tree node. Otherwise, suppose that the theorem is true for $1, 2, \ldots, r$. Consider a leftist tree with $r + 1$ nodes on the right path. Then the root has a right subtree with r nodes on the right path, and a left subtree with at least r nodes on the right path (otherwise it would not be leftist). Applying the inductive hypothesis to these subtrees yields a minimum of $2^r - 1$ nodes in each subtree. This plus the root gives at least $2^{r+1} - 1$ nodes in the tree, proving the theorem.

LEFTIST HEAP

- In summary, the merge of the leftist heap will take O(log(n)) time.
- The element with the highest priority will reside on the root, and retrieving that element will only take O(1) time.
- enqueue() can be done through merging the existing leftist heap with a leftist heap that contains only the element to be inserted. dequeue() can be done through deleting the root and subsequently merging the remaining left and right subtrees. Both operations will take O(log(n)) time.
- Unfortunately, unlike the regular heap, leftist heap cannot be implemented using array (because the leftist heap is not full). We will need to use pointers for the implementation, which is less computational and space efficient. **So, only use leftist heap if merge is a frequent operation**.

BINOMIAL QUEUE

- Binomial queue is another data structure that supports fast merge operation.

- It still requires O(1) time for retrieving the element with the highest priority, and O(log(n)) for merge, enqueue(), and dequeue() in the worst-case scenario.
  - but it only requires, on average, constant time for enqueue() (this is better than the leftist heap)

BINOMIAL QUEUE

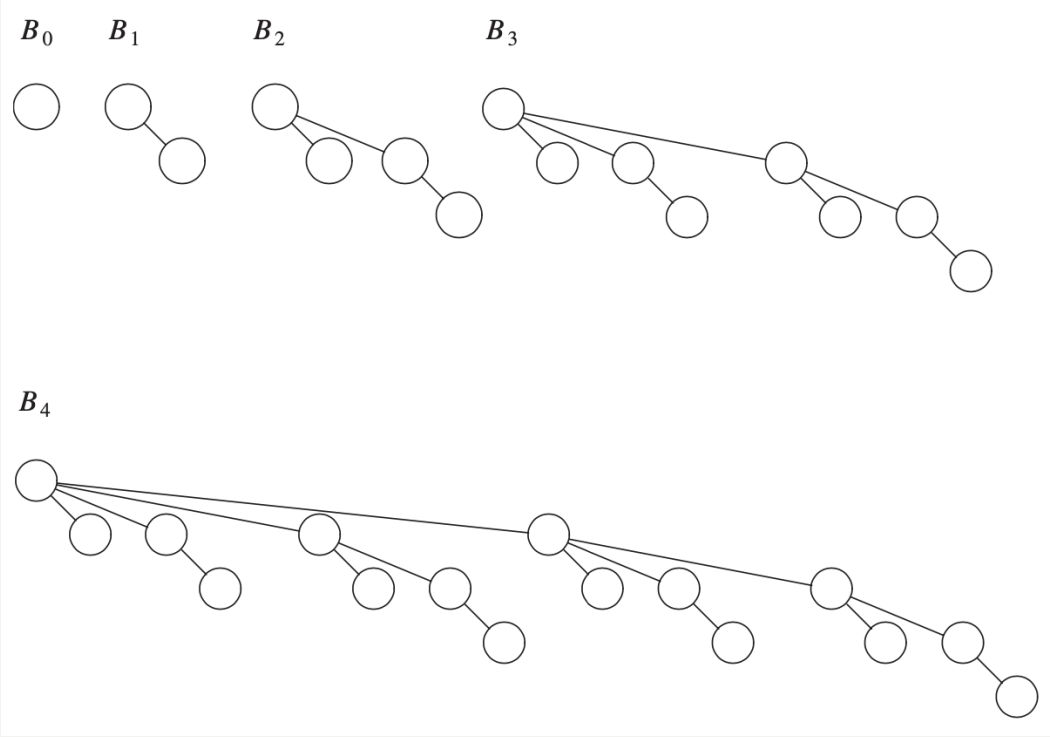
- A binomial queue is a set of trees (also called **forest**).

- The trees contain different number of nodes. The different numbers of nodes contained in different trees correspond to the different exponents of 2. For example, there could be a tree that contains 1 node (2^0), another tree that contains 16 nodes (2^4), another tree that contains 64 nodes (2^6)… However, we will not have any tree that contains, e.g., 6 nodes, which is not an exponent of 2. Hence, each tree is called a **binomial tree**. (And it is also where the name **binomial queue** comes from.)

- The sets of trees to use depends on the number of elements in the queue, which is broken down into the sum of exponents of 2 that will be used to determine the sets of tree in use. For example, if we have 381 nodes, it can be broken down into 1+4+8+16+32+64+256. So, we will use a tree with 1 node, a tree with 4 nodes, a tree with 8 nodes, …, and a tree with 256 nodes.

BINOMIAL QUEUE

- The trees are defined recursively. Let B_i be the tree that contains 2^i nodes. B_0 is a one-node tree. B_k is formed by attaching a binomial tree B_{k-1} to the root of another binomial tree B_{k-1}.
- Each binomial tree has the heap property: the parent is smaller (with a higher priority) than its children.



BINOMIAL QUEUE: OPERATIONS

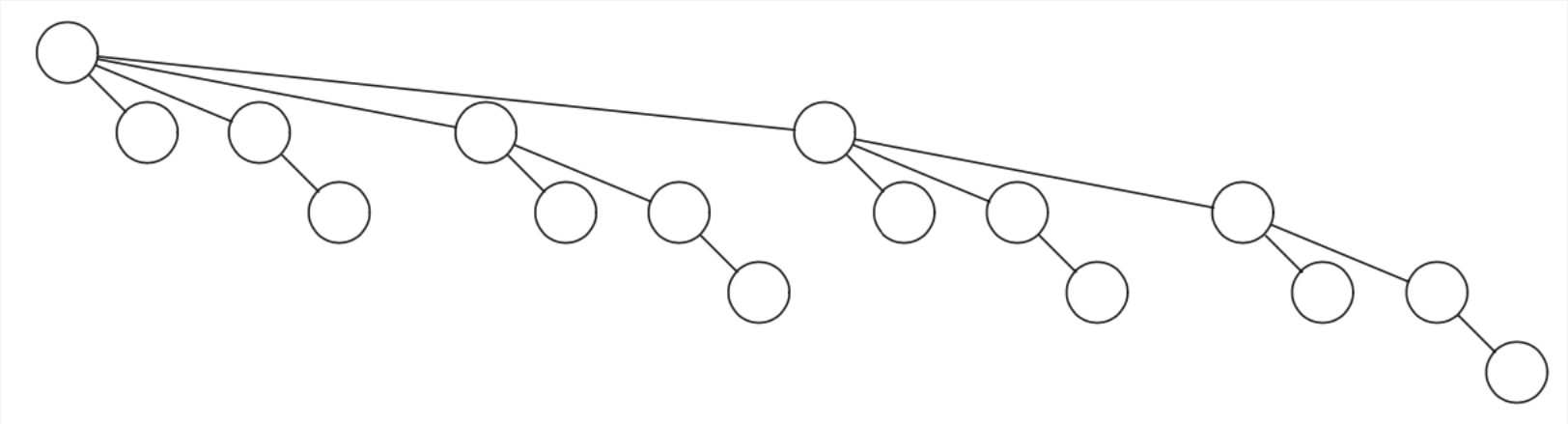
- We claim that we can access the element with the highest priority in O(1) time. To achieve this, we can simply set up a pointer that points to the smallest root among all binomial trees. (Keep in mind that we will need to update this pointer when insertion/deletion happens.)

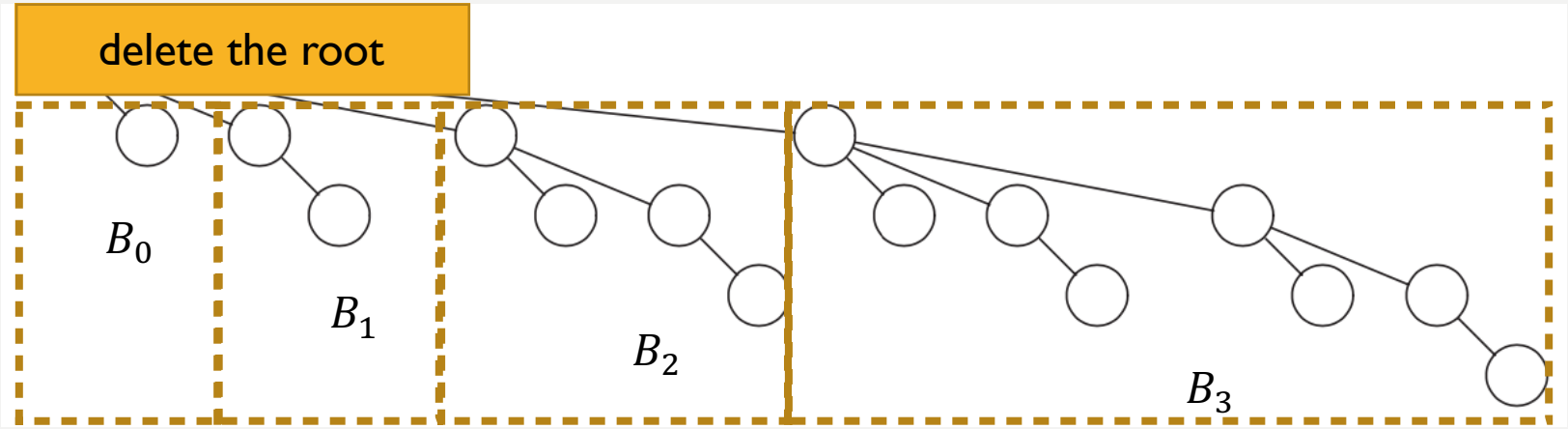
- Similar to leftist heap, enqueue() and dequeue() of the binomial queue can also be performed through using merge():
  - enqueue(): merge the existing binomial queue with a binomial queue that contains a single node
  - dequeue(): deleting the root of a binomial tree will form a binomial queue, we can than merge the existing binomial queue (without the binomial tree under operation) and the newly formed (by deleting the node) binomial queue

BINOMIAL QUEUE: OPERATIONS

- Deletion: example

B_4





BINOMIAL QUEUE: OPERATIONS

- Similar to leftist heap, binomial heap also takes merge as its essential operation.
- The merge process of binomial queue resembles simple arithmetic addition of two binary numbers.

BINOMIAL QUEUE: OPERATIONS

- For example, let's say we have two binomial queues, one with 15 nodes and another with 12 nodes. We can decompose the first queue (say A) as 1+2+4+8 (or $B_0^A + B_1^A + B_2^A + B_3^A$), and the second queue (say B) as $B_2^B + B_3^B$.

- The merge is essentially looking to construct a new queue $C = A + B = B_0^A + B_1^A + B_2^A + B_3^A + B_2^B + B_3^B$.

BINOMIAL QUEUE: OPERATIONS

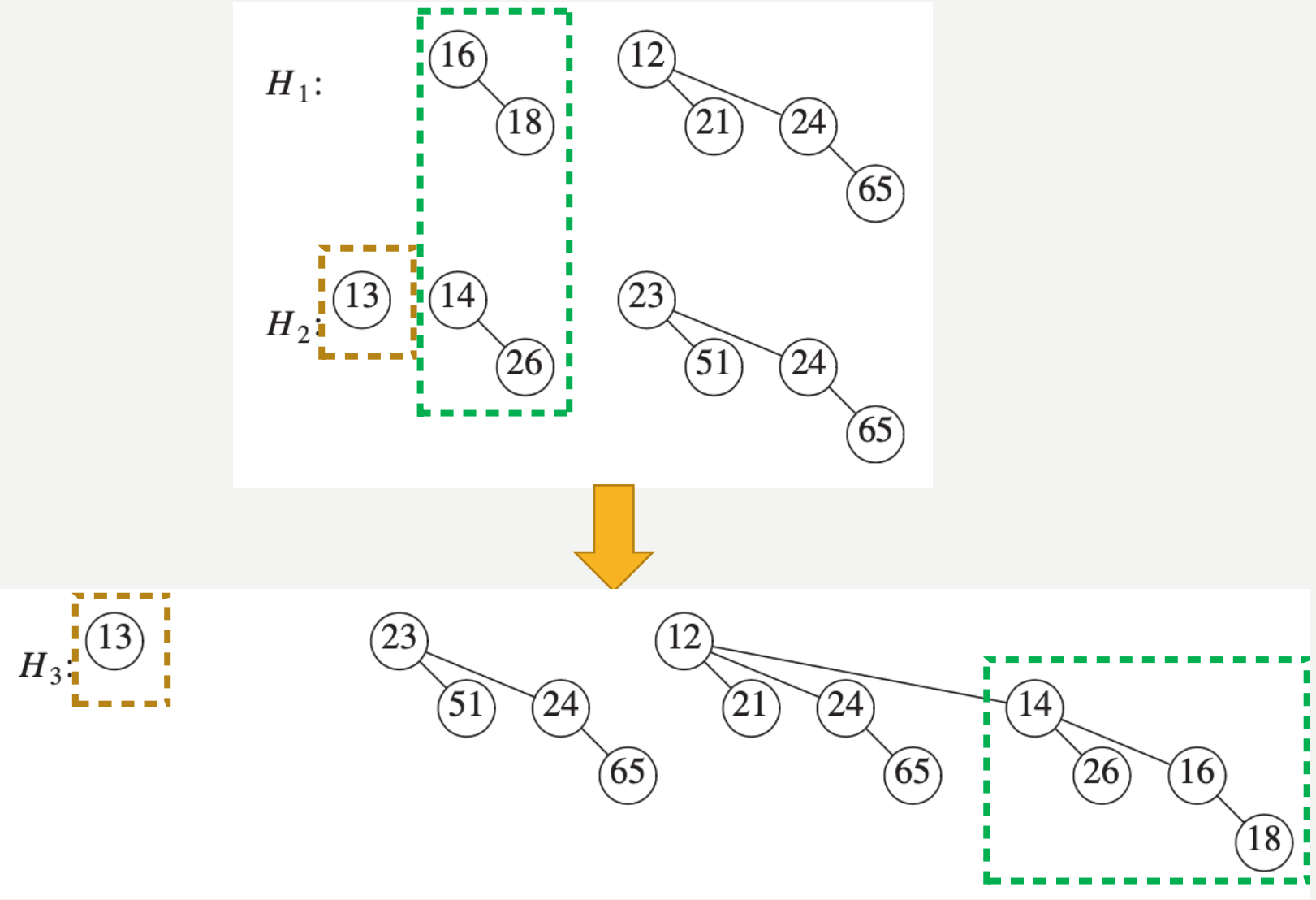
- Recall the recursive definition of binomial tree: it says that we can merge two B_{k-1}s to get a new B_k. **To retain the heap property, we attach the binomial tree with a larger root the the one with a smaller root**.

- Let's go back to the original problem of computing $C = B_0^A + B_1^A + B_2^A + B_3^A + B_2^B + B_3^B$.

- Starting from the smallest trees, we notice that we only have one B_0, and in this case we will directly take this tree, B_0^A, into our new queue ($B_0^A = B_0^C$). Similarly, we also take B_1^A directly ($B_1^A = B_1^C$). Next, we notice that we have two B_2s, and we will merge them into one B_3, say B_3^C. Finally, we also have two B_3s and we will merge them into one B_4, say B_4^C.

- Finally, we will have $C = B_0^C + B_1^C + B_3^C + B_4^C$. We can double check the total number of elements in the new queue, we have 1+2+8+16=27 elements, which equals to the total number of elements in the original queues (27=15+12).

BINOMIAL QUEUE: OPERATIONS



BINOMIAL QUEUE: OPERATIONS

- Merge time complexity: O(log(n))
  - we know that the binomial queue that contains n elements has log(n) binomial trees, because each tree is at least twice larger than the previous tree
  - merging each pair of binomial trees only takes O(1) time (specifically, one root comparison, and one attachment)

- Because enqueue() and dequeue() can be done through a single call of the merge function, both of them will also require O(log(n)) time.

BINOMIAL QUEUE: LINEAR-TIME INSERTION

- Remember at the beginning, we claim that enqueue() can be done in O(1) time on average. However, our previous analysis shows that it needs O(log(n)) time in the worst case scenario.

- To see the O(1) time average complexity, we need to model the merge process as a Bernoulli process. Just like flipping a coin, we consider that the chance the existing binomial queue contains a binomial tree with a specific size as a random variable with 50-50 chance. That is, if we do not know the the size of the binomial queue, we will assume that it has 50% chance to contain B_0, 50% chance to contain B_1,… etc.

BINOMIAL QUEUE: LINEAR-TIME INSERTION

- Consider the enqueue() process, where we recursively try to merge the single-node binomial tree with the element to insert with the existing binomial trees. The merge process will stop if the existing binomial queue does not contain a specific binomial tree.

- For example, consider a binomial queue $B_0 B_1 B_3 B_4$, the insertion will terminate after two steps because the existing tree does not contain B_2.

- The number of steps we need for the insertion thus equals to the number of continuously-double-sized binomial trees that the queue has.

BINOMIAL QUEUE: LINEAR-TIME INSERTION

- If we consider that the existing binomial queue contains a specific binomial tree as getting a head in a coin-flip (both has 50% chance), then the number of merge steps we need will correspond to the number of heads we can get before getting a tail.

- The distribution of this random variable is modeled by the **negative binomial distribution**.

- The negative binomial distribution has an expected value of $p/(1-p)$, where p is the chance of getting a head in the experiment. In our case, $p = 0.5$ and the expected value is 1. It says we expect to see one head before seeing one tail (which makes a general sense for a fair coin toss).

- Therefore, we can expect to only trigger 1 merge operation for each insertion.

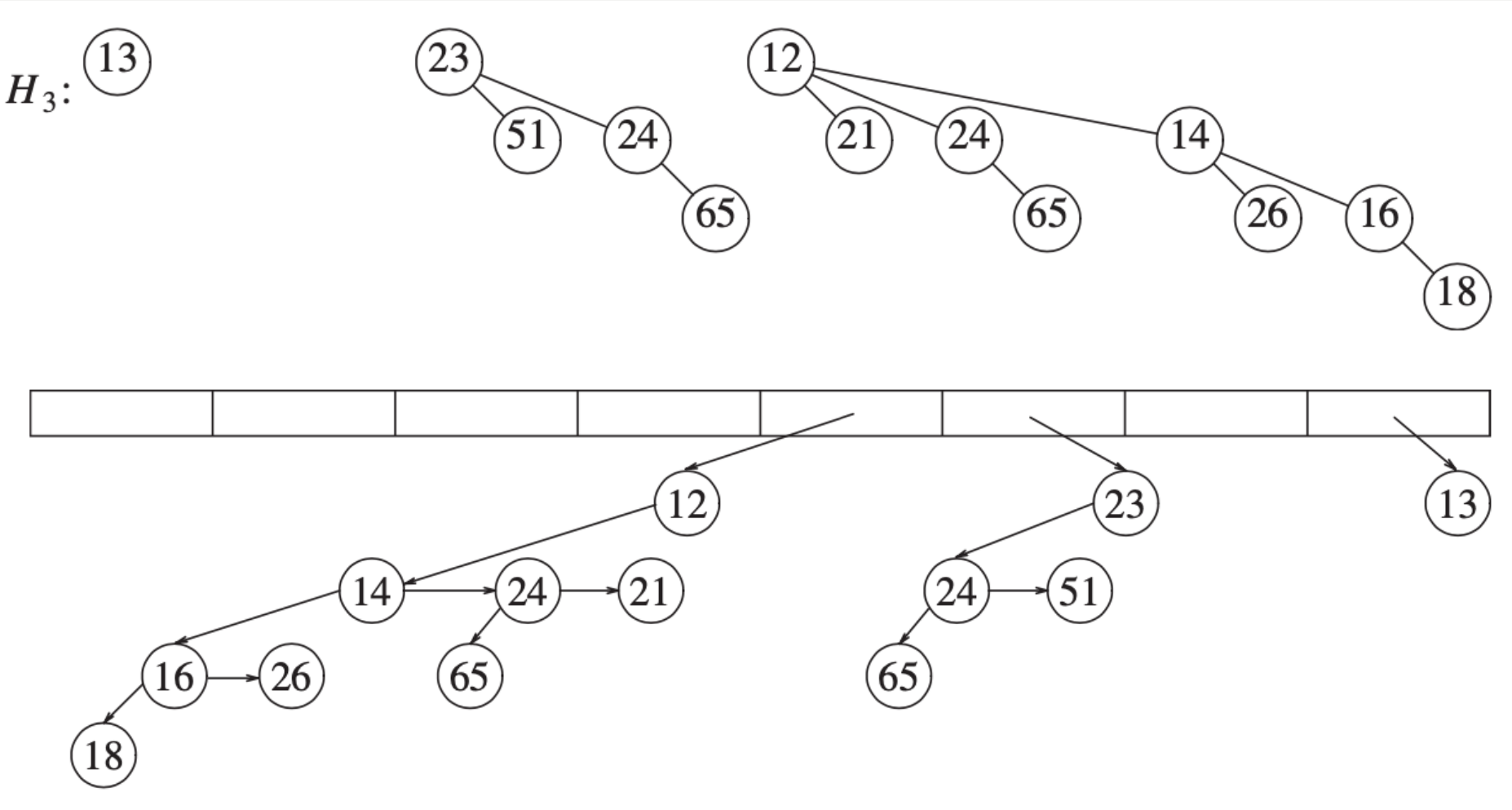
BINOMIAL QUEUE: SUMMARY

- Binomial queue contains a set of binomial trees, each with different sizes that are exponents of 2.

- A binomial queue with n elements contains O(log(n)) binomial trees. This is the fundamental observation to establish its related time complexity.

- Binomial queue supports:
  - top(): O(1)
  - enqueue: O(log(n)) in the worst-case scenario, O(1) on average (note that when we merge, we also need to update the MIN pointer if necessary; the MIN pointer helps us to locate the element with the highest priority in a constant time)
  - dequeue(): O(log(n))
  - merge(): O(log(n))

BINOMIAL QUEUE: IMPLEMENTATION



BINOMIAL QUEUE: IMPLEMENTATION

```
26      private:
27        struct BinomialNode
28        {
29            Comparable    element;
30            BinomialNode *leftChild;
31            BinomialNode *nextSibling;
32
33            BinomialNode( const Comparable & e, BinomialNode *lt, BinomialNode *rt )
34              : element{ e }, leftChild{ lt }, nextSibling{ rt } { }
35
36            BinomialNode( Comparable && e, BinomialNode *lt, BinomialNode *rt )
37              : element{ std::move( e ) }, leftChild{ lt }, nextSibling{ rt } { }
38        };
39
40        const static int DEFAULT_TREES = 1;
41
42        vector<BinomialNode *> theTrees;  // An array of tree roots  // the array containing pointers to the subtrees
43        int currentSize;                  // Number of items in the priority queue
```

BINOMIAL QUEUE: IMPLEMENTATION

```
1       /**
2        * Return the result of merging equal-sized t1 and t2.
3        */
4       BinomialNode * combineTrees( BinomialNode *t1, BinomialNode *t2 )
5       {
6           if( t2->element < t1->element )
7               return combineTrees( t2, t1 );
8           t2->nextSibling = t1->leftChild;
9           t1->leftChild = t2;
10          return t1;
11      }
```

// compare the root; attach the subtree with a larger root to the one with a smaller root to maintain the heap property

BINOMIAL QUEUE: IMPLEMENTATION

```
22          BinomialNode *carry = nullptr;
23          for( int i = 0, j = 1; j <= currentSize; ++i, j *= 2 )
24          {
25              BinomialNode *t1 = theTrees[ i ];
26              BinomialNode *t2 = i < rhs.theTrees.size( ) ? rhs.theTrees[ i ]
27                                                          : nullptr;
28              int whichCase = t1 == nullptr ? 0 : 1;   // whether the first queue contains the subtree
29              whichCase += t2 == nullptr ? 0 : 2;      // whether the second queue contains the subtree
30              whichCase += carry == nullptr ? 0 : 4;   // whether exists a carrying-over subtree
```

BINOMIAL QUEUE: IMPLEMENTATION

```
32              switch( whichCase )
33              {
34                case 0: /* No trees */
35                case 1: /* Only this */
36                  break;
37                case 2: /* Only rhs */
38                  theTrees[ i ] = t2;
39                  rhs.theTrees[ i ] = nullptr;
40                  break;
41                case 4: /* Only carry */
42                  theTrees[ i ] = carry;
43                  carry = nullptr;
44                  break;
45                case 3: /* this and rhs */
46                  carry = combineTrees( t1, t2 );
47                  theTrees[ i ] = rhs.theTrees[ i ] = nullptr;
48                  break;
49                case 5: /* this and carry */
50                  carry = combineTrees( t1, carry );
51                  theTrees[ i ] = nullptr;
52                  break;
53                case 6: /* rhs and carry */
54                  carry = combineTrees( t2, carry );
55                  rhs.theTrees[ i ] = nullptr;
56                  break;
57                case 7: /* All three */
58                  theTrees[ i ] = carry;
59                  carry = combineTrees( t1, t2 );
60                  rhs.theTrees[ i ] = nullptr;
61                  break;
62              }
63          }
```

C++ STL PRIORITY QUEUE

- std::priority_queue
  - void push(const Object & x);
  - const Object & top( ) const;
  - void pop( );
  - bool empty( );
  - void clear( );

SUMMARY

- Priority queue: an extension of the queue ADT with user-defined feature to determine priority

- Implementations
  - heap: array implementation, fast and simple, but does not support merge well (needs O(n)) time for merge
  - leftist heap: supports O(log(n)) merge, but requires pointers that makes it slower and more memory-intensive
  - binomial queue: also supports O(log(n)) merge, further improves leftist heap with an expected O(1) enqueue() time.