CSCI 335 Software Design and Analysis III
Lecture 16
Priority Queues:
Skew, Leftist heaps, Binomial Queues

Professor Anita Raja

10-31-22

Annoucements

- HW2 grades released
- HW3 in process
- Midterm grades released later this week.
- Some changes so far based on survey
  - Discussion of code and assignment in class
  - Full sized slides
  - Slides clearly only a subset of the material discussed in class.
    - They serve as helpful guide/reference for course material.
  - More frequent low stake assessment: this week pop quiz on hask tables and heaps.

Agenda

- buildHeap, proof, selection problem, dheap
- Skew Heap
- Leftist Heap
- Binomial Queues

Leftist Heap

- Supports o(N) merge, in particular logarithmic
- **Ordering property**: as in regular heaps
- **Structural property**: not enforced explicitly
  - Null path length of node X: **npl(X)** = length of shortest path from node to a node without 2 children

$$npl(X) = 1 + \min\{npl(children)\}$$

  - npl(empty tree)=-1

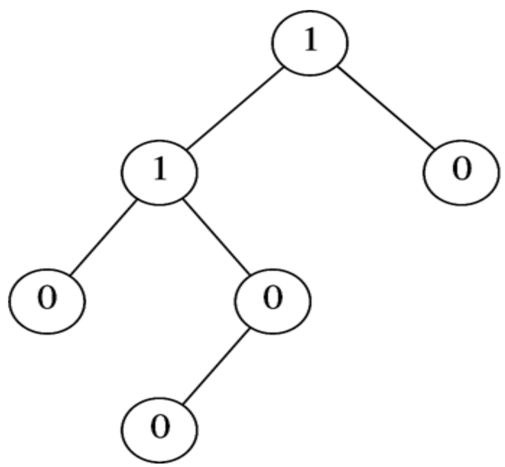
In leftist heaps (definition):

For every node X, the npl() of left child is greater or equal to npl() of right child, therefore:

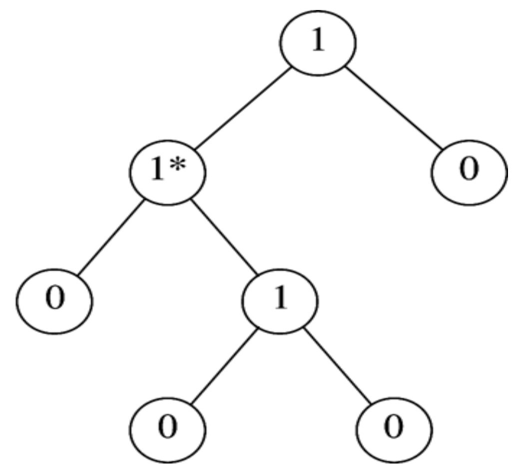
npl(X)=1+npl(right child)

Leftist heaps

npl() of nodes shown



Leftist



not leftist

Leftist heaps: implementation

- Keep value of npl() at each node.
- Update npl() as necessary.

Theorem

- **Theorem:** A leftist tree with r nodes on the right path must have at least 2^r -1 nodes [**Proof by induction]**
- Therefore a leftist tree with N nodes total,

  and exactly r nodes on right path has r $\leq$ log(N+1) nodes on right path
- So, right path is short.

=> Merge based on the right path only.

Merge leftist heaps H1, H2

- Recursive algorithm:

(1) If one of them is empty return the other one (basis)

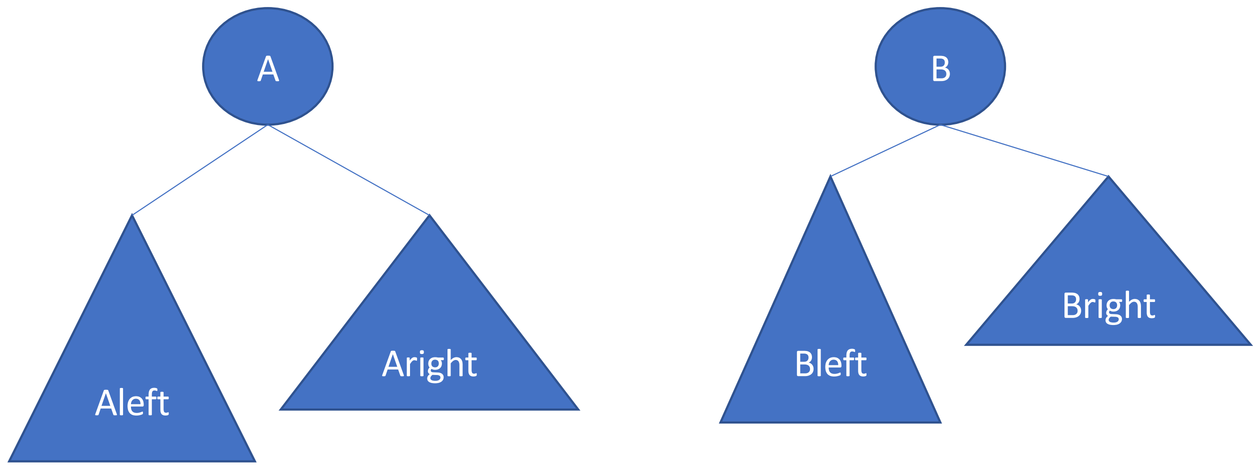
Otherwise,

(2) recursively merge heap with larger root, with the right subheap of the tree with the smaller root. [**Result is a leftist heap**]

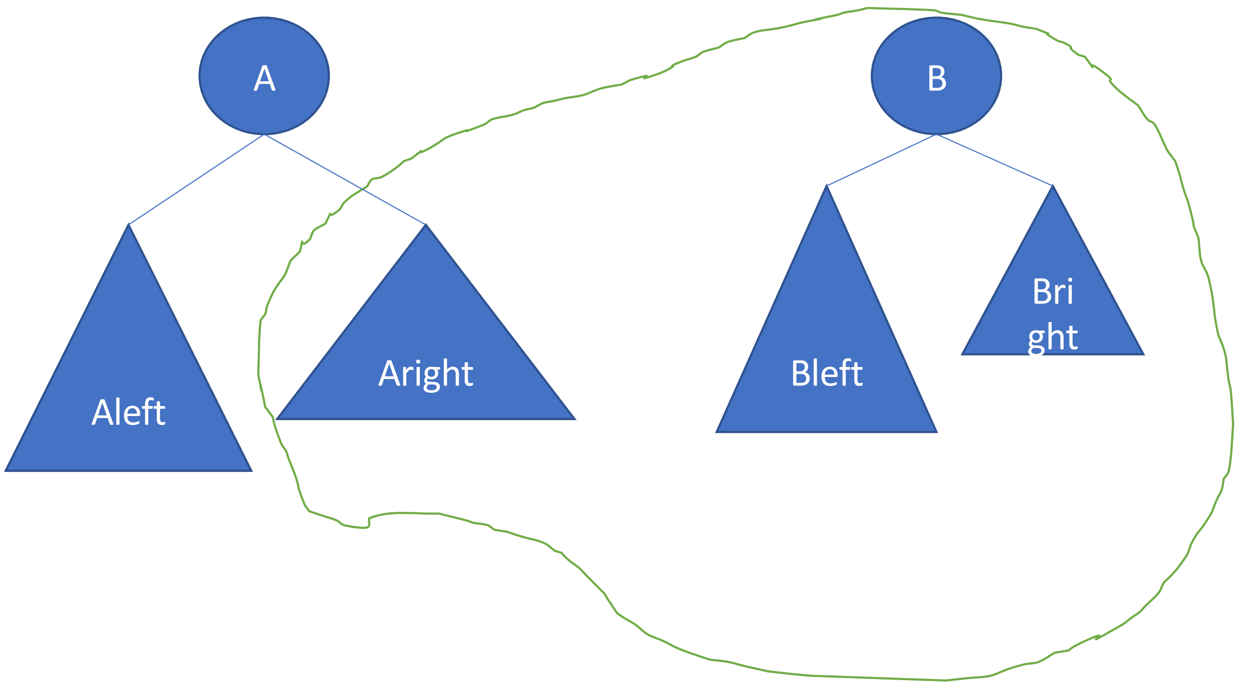
(3) finally make the right child of the heap with the smaller root, the one resulted from step (2)

<Plus: leftist heap restoration via simple swap if needed>

Two leftist heaps, A < B

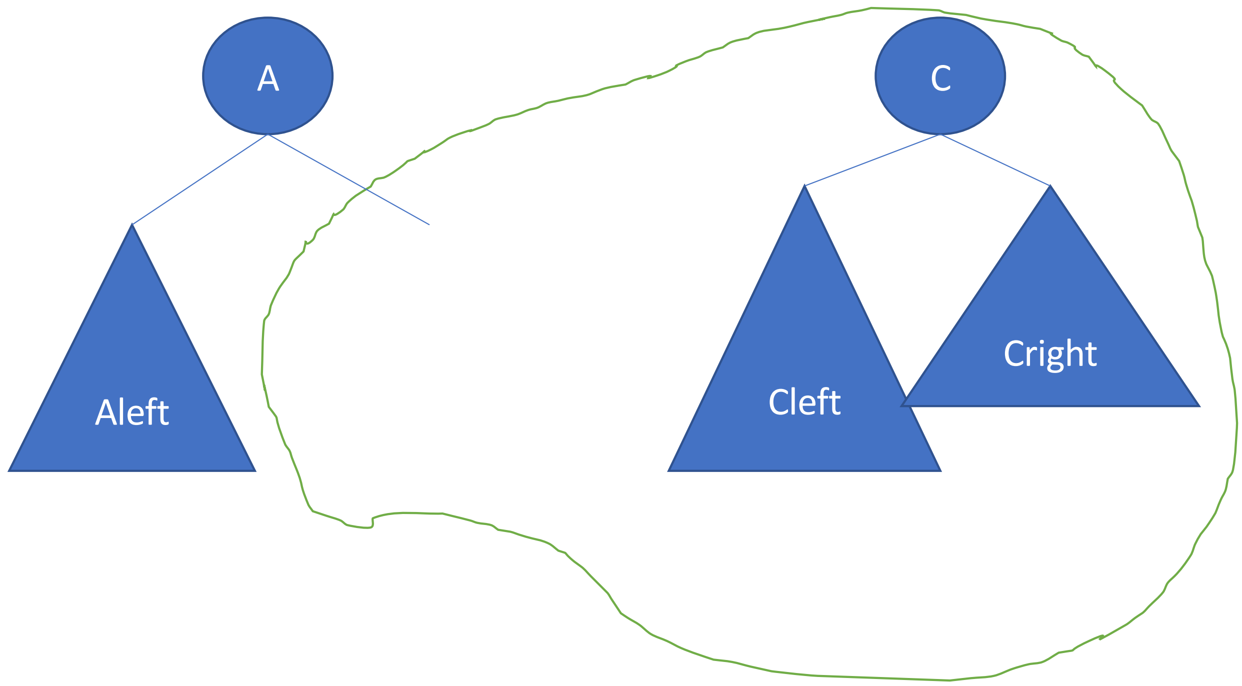


Two leftist heaps, A < B



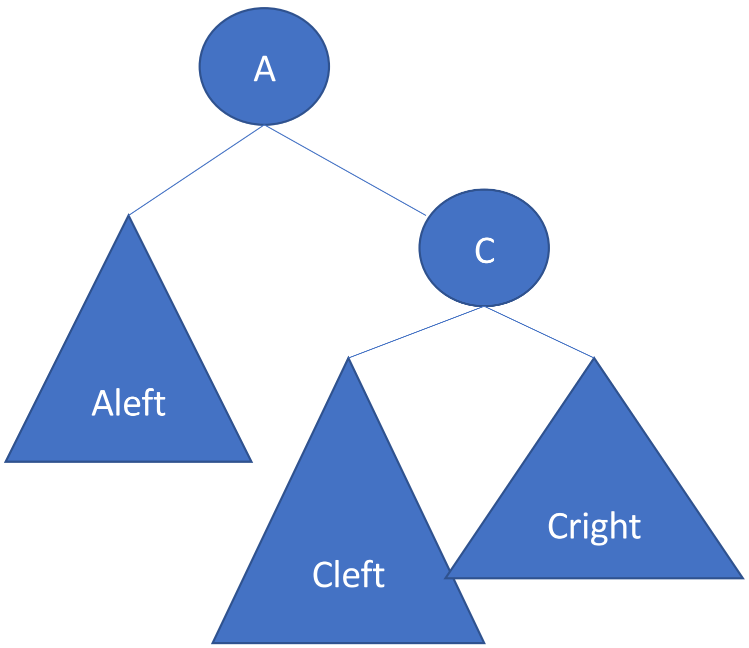
MERGE

Two leftist heaps, A < B



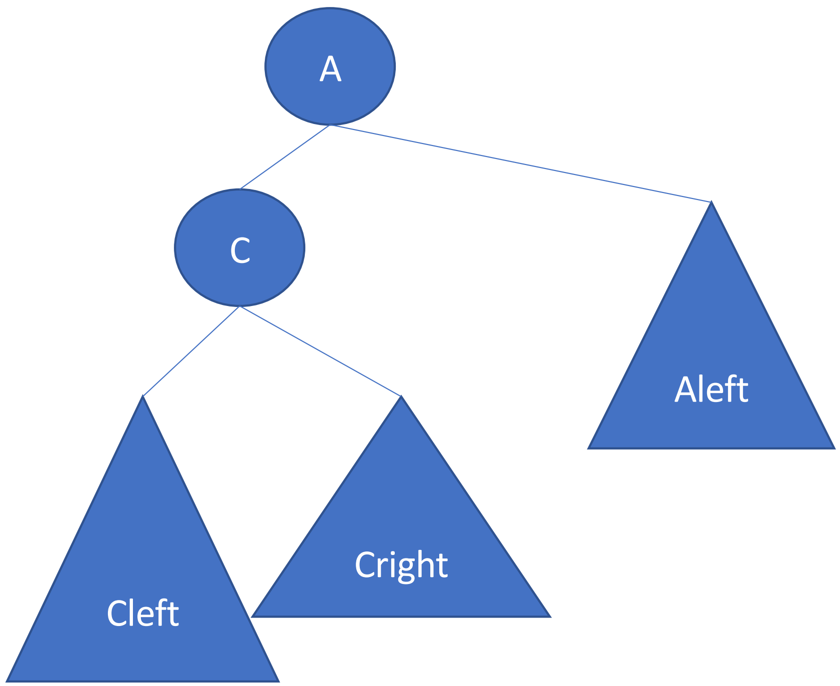
Result of Merge is a leftist heap

Two leftist heaps, A < B



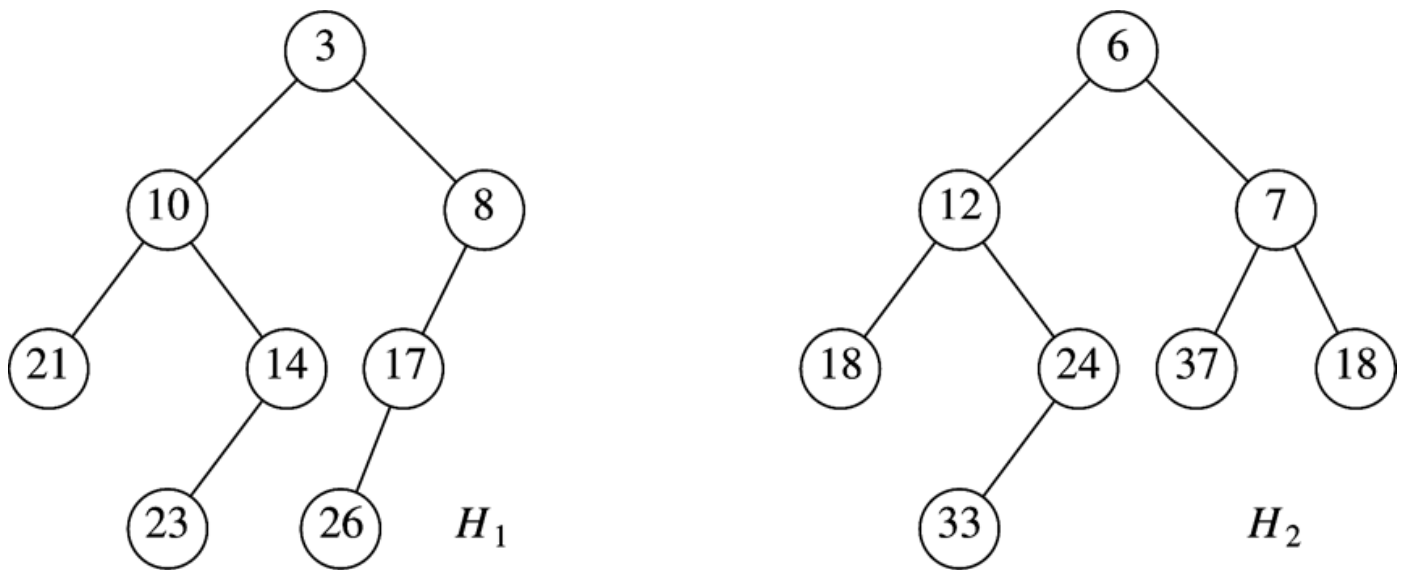
Attach result as right child of A

Two leftist heaps, A < B (root)



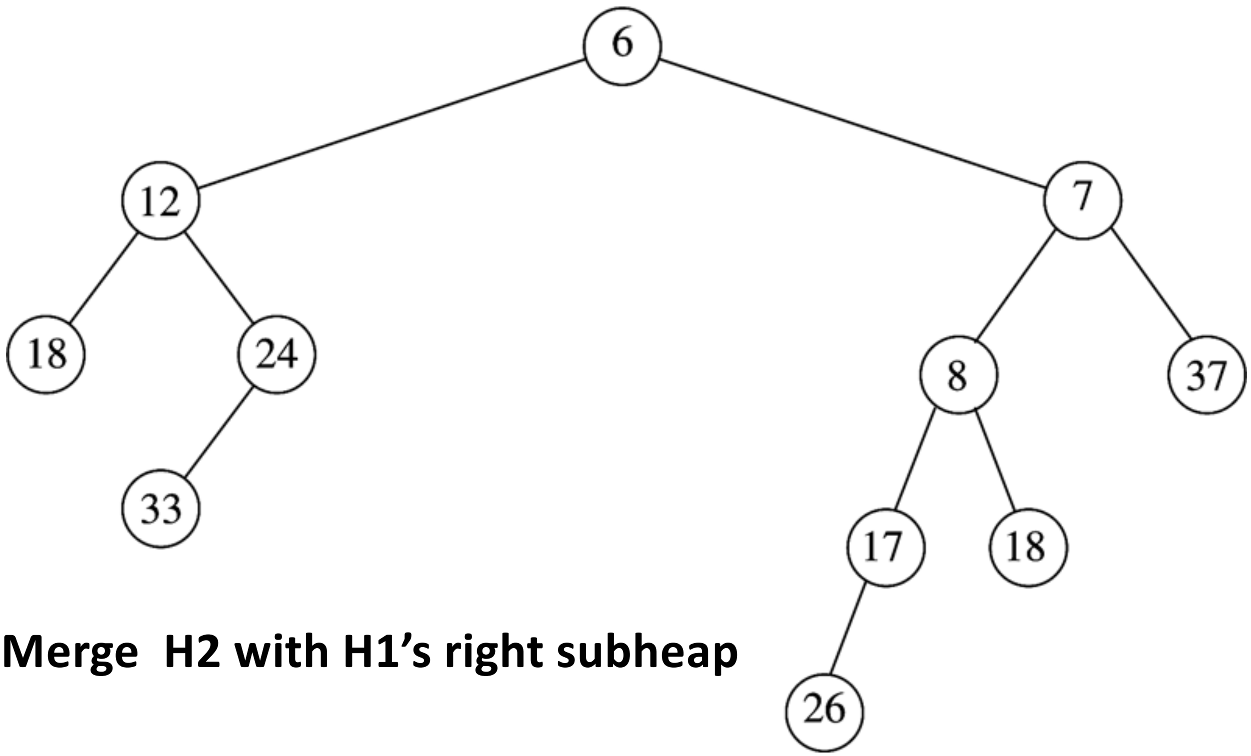
Swap children if needed. Update npl() of root

Merge leftist heaps



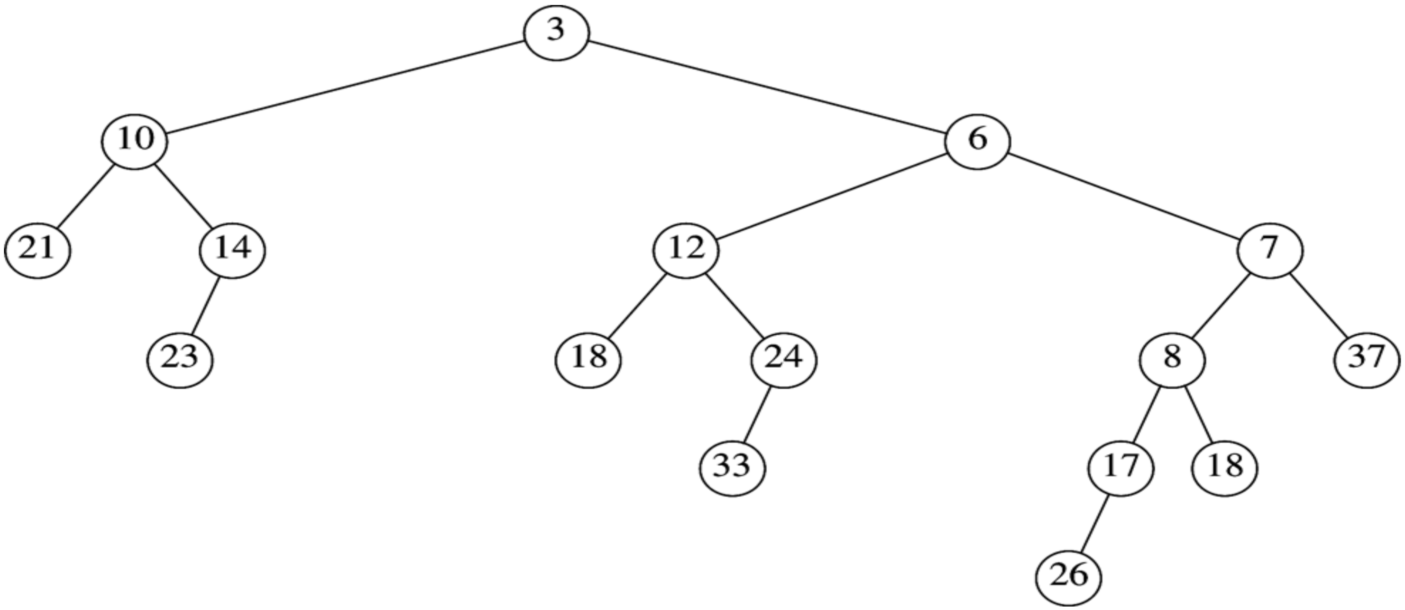
Input

Merge leftist heaps



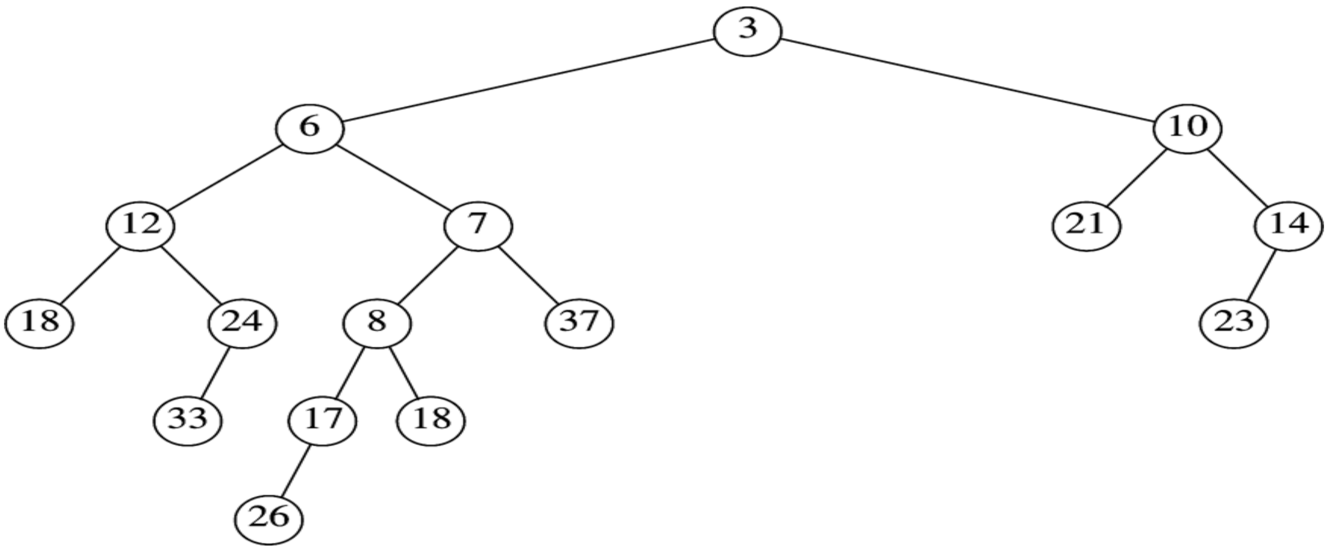
(2) Merge H2 with H1's right subheap

Merge leftist heaps



(3) Attach previous result as right child of H1

Merge leftist heaps



(3) Restore leftist property by swapping children under root
The result of (2) is a leftist heap

Code

```
void merge(LeftistHeap &rhs) {
    if (this == &rhs)    // Avoid aliasing problems
      return;
    root_ = Merge(root_, rhs.root_);
    rhs.root_ = nullptr;
}

LeftistNode *Merge(LeftistNode *h1, LeftistNode *h2)
{
    if (h1 == nullptr)      // Base cases
        return h2;
    if (h2 == nullptr)
        return h1;
    if (h1->element_ < h2->element_)
        return Merge1(h1, h2);
    else
        return Merge1(h2, h1);
}
```

Base case

Recursive merge

Code

Base case (other)

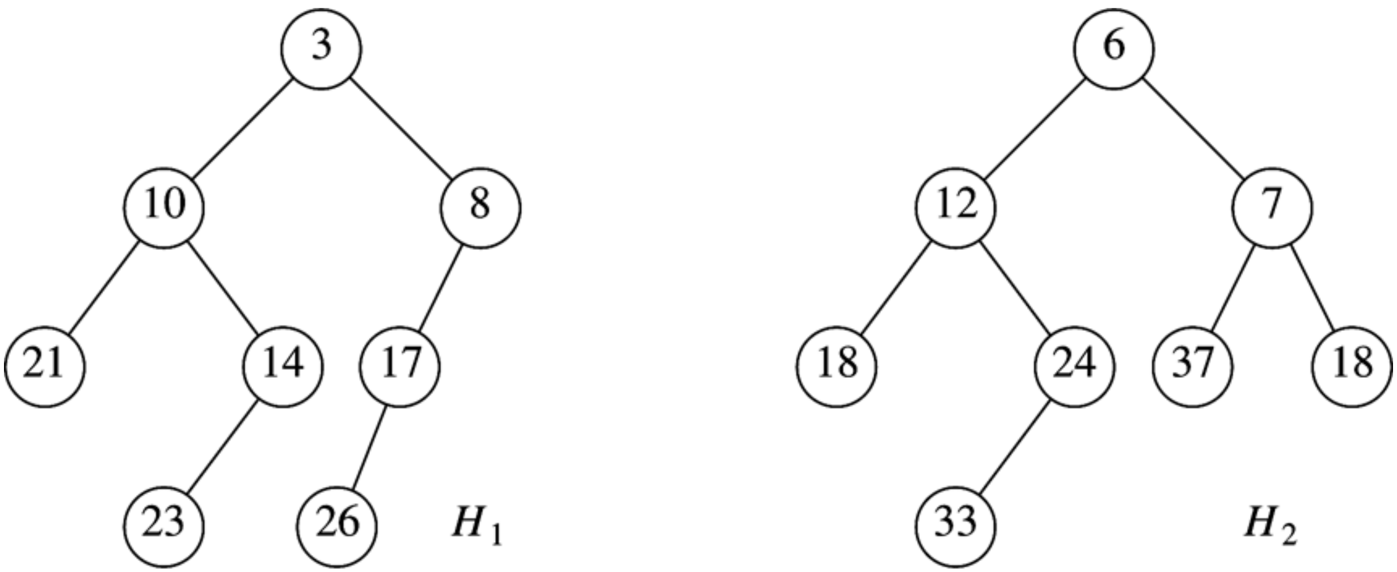
Recursive merge

Make leftist

Update npl

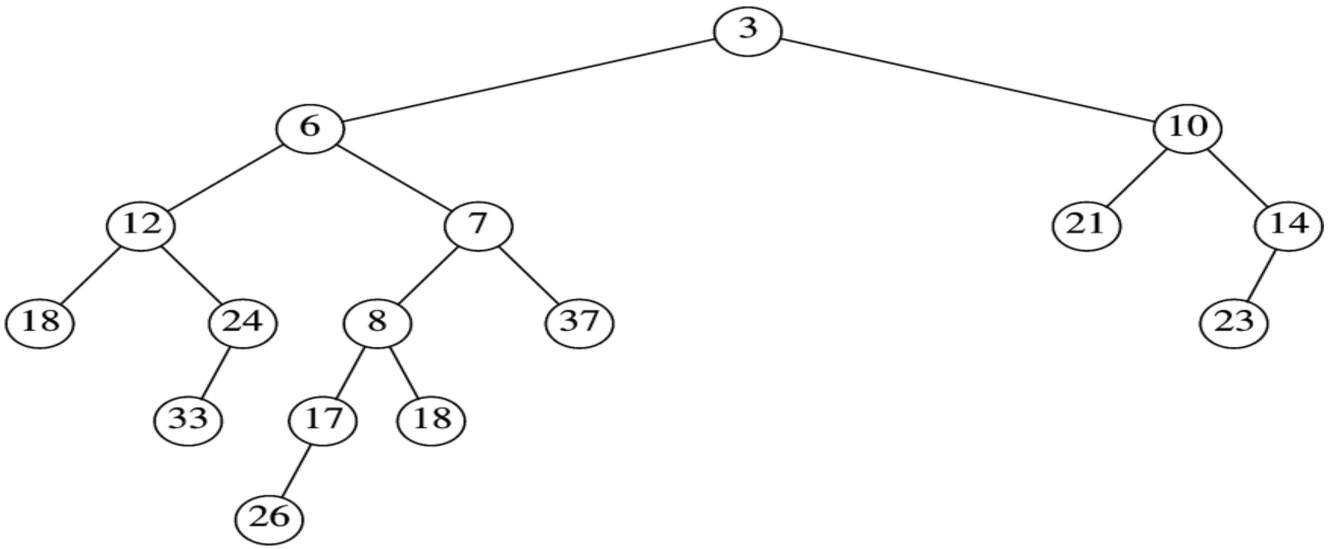
```
/**
 * Internal method to merge two roots.
 * Assumes trees are not empty, and h1's root contains smallest item.
 */
LeftistNode * merge1( LeftistNode *h1, LeftistNode *h2 )
{
    if( h1->left == NULL )    // Single node
        h1->left = h2;        // Other fields in h1 already accurate
    else
    {
        h1->right = merge( h1->right, h2 );
        if( h1->left->npl < h1->right->npl )
            swapChildren( h1 );
        h1->npl = h1->right->npl + 1;
    }
    return h1;
}
```

Merge leftist heaps



Input

Final Result



Merge leftist Heaps

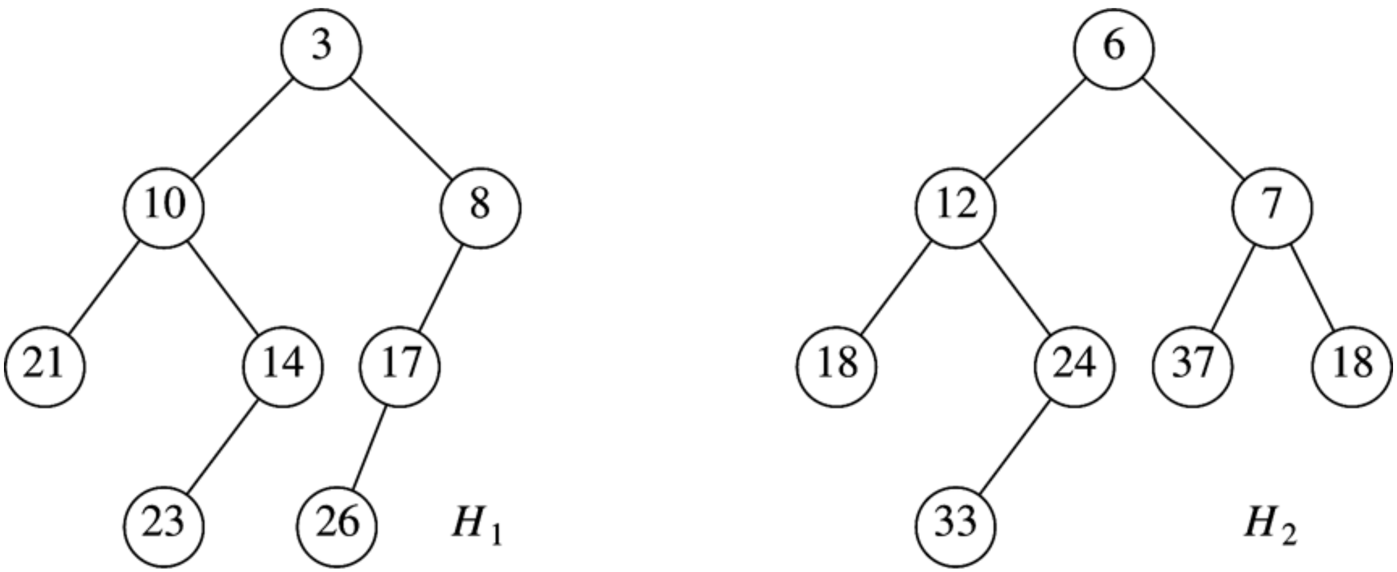
- Non-recursive implementation:

Pass 1) Arrange nodes of right paths of H1, and H2 in sorted order, keeping their respective left children

Pass 2) Bottom-up -> swap children that violate leftist property

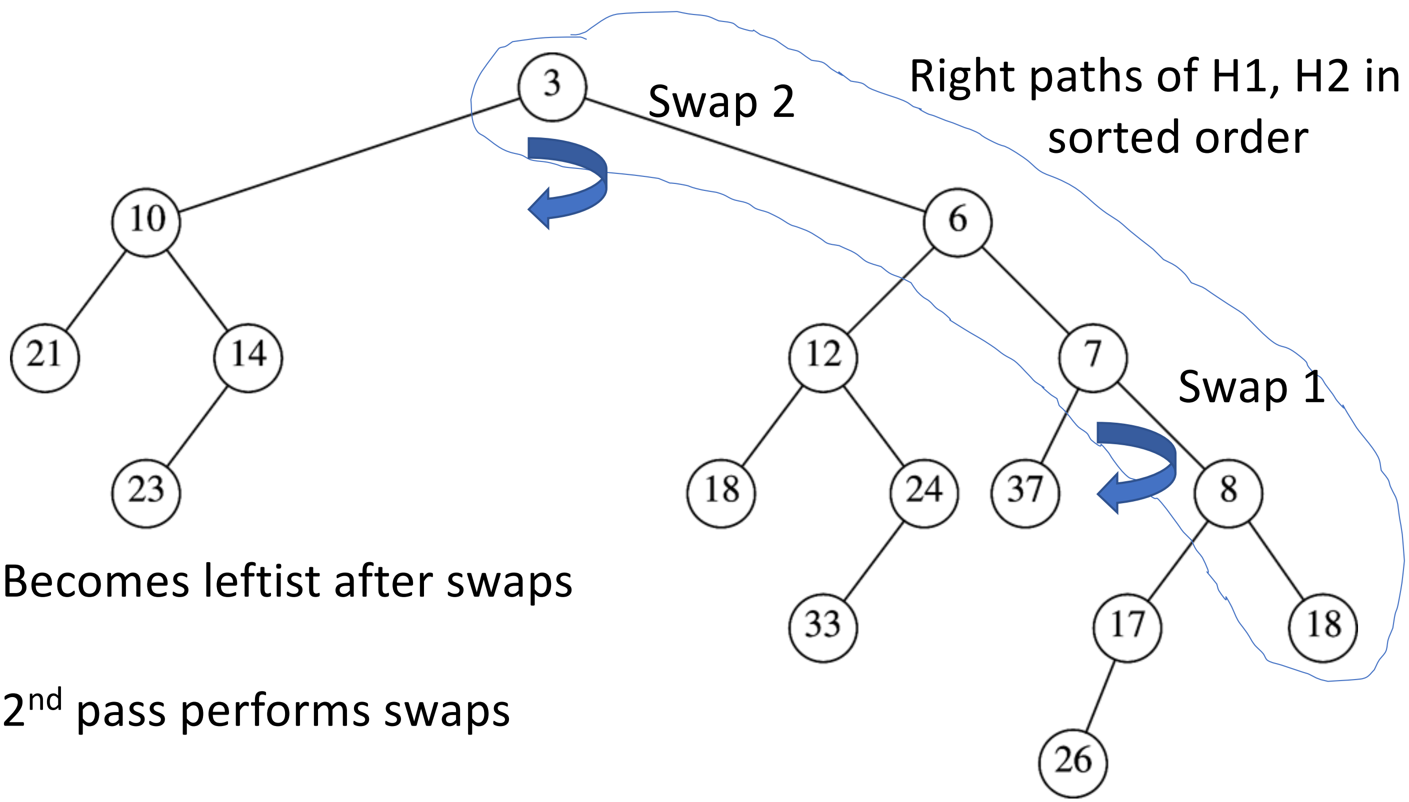
Simple to view

Merge leftist heaps



Input

Non-recursive merge (first pass)



Becomes leftist after swaps

2nd pass performs swaps

Summary (leftist heaps)

- Merge is an O(logN) operation
- Insert, DeleteMin ?

Summary (leftist heaps)

- Merge is an O(logN) operation
- Insert, deleteMin ?
  - Insert: Merge current heap with a single-node heap
  - deleteMin: Destroy root, merge two subheaps
- So Insert, deleteMin: O(logN)

Skew Heaps

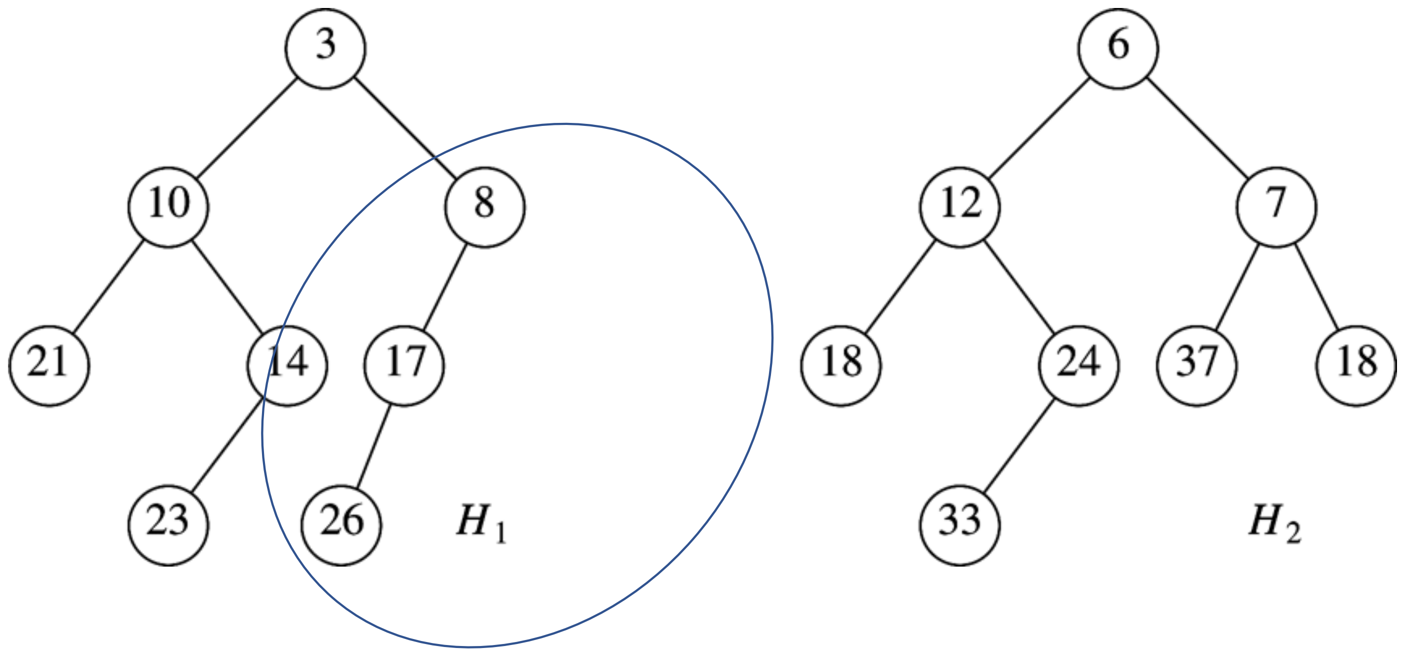
- Self-adjusting heaps

  Leftist heaps vs. skew heaps

  Avl trees vs. splay trees

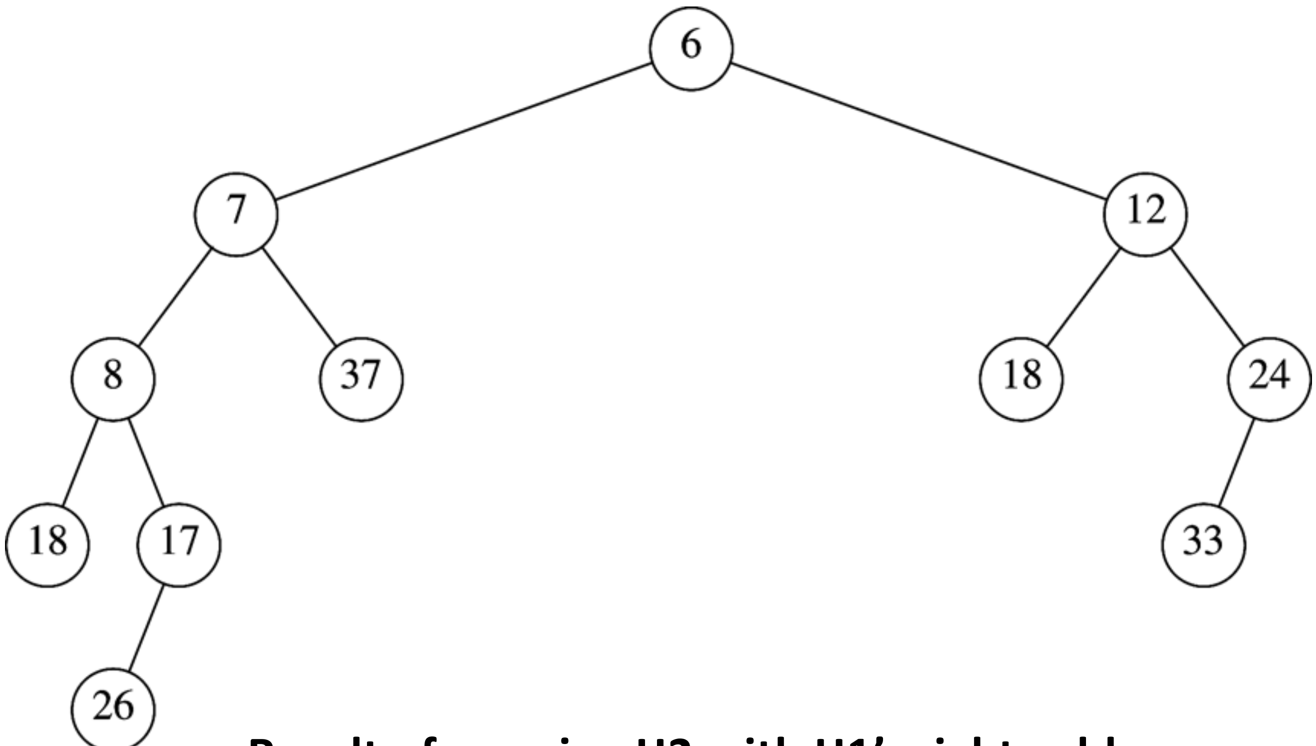
- O(logN) amortized cost
- Merge similar as in leftist heaps
  - No need to keep npl() information
  - Always swap children with one exception
    - Exception: Never swap children of largest node on right path.

Example



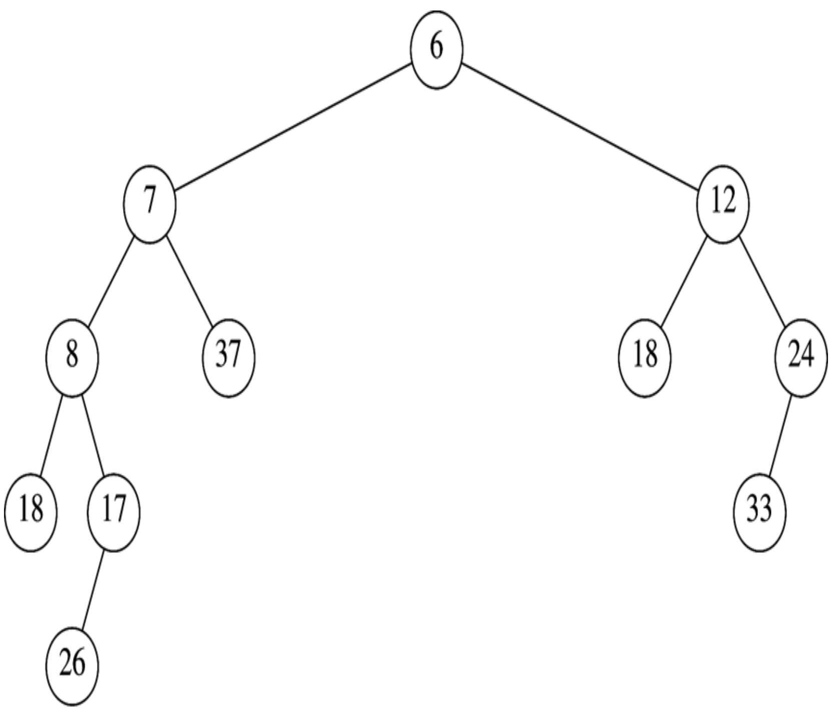
Merge skew heaps

Result

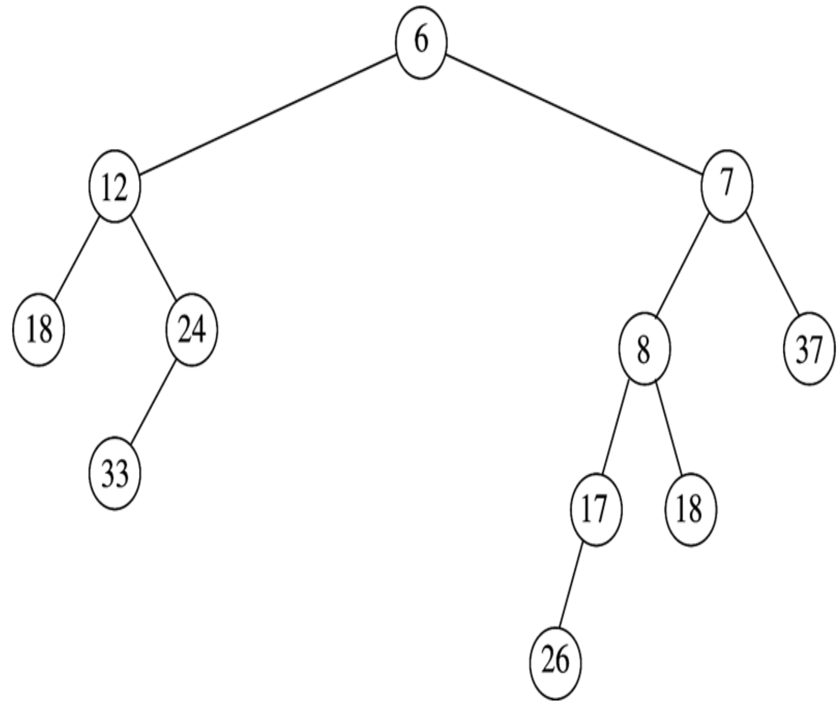


Result of merging H2 with H1's right subheap

Compare w/ leftist heaps

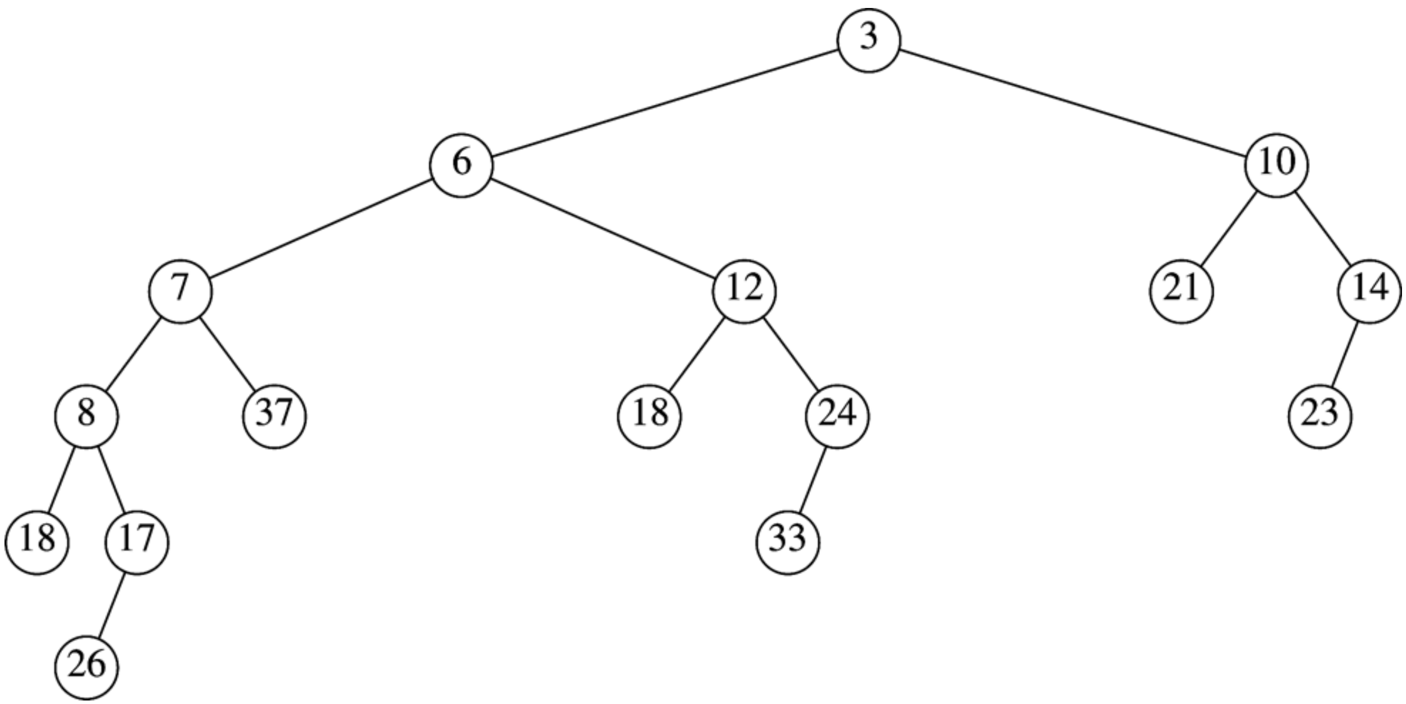


(2) Skew : Merge H2 with H1’s right subheap

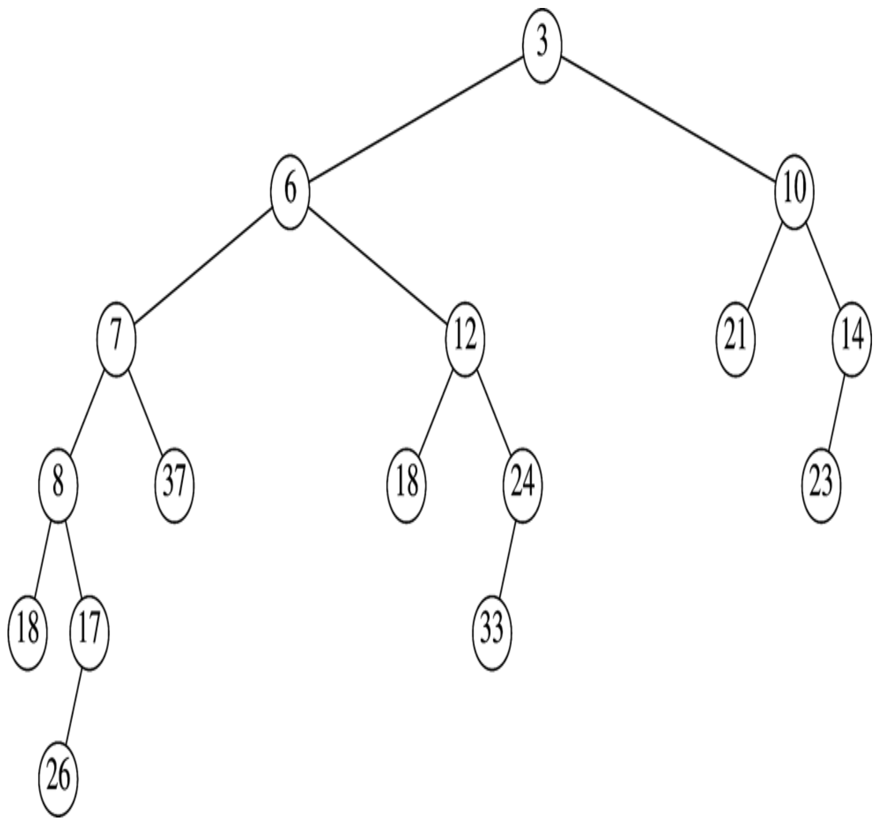


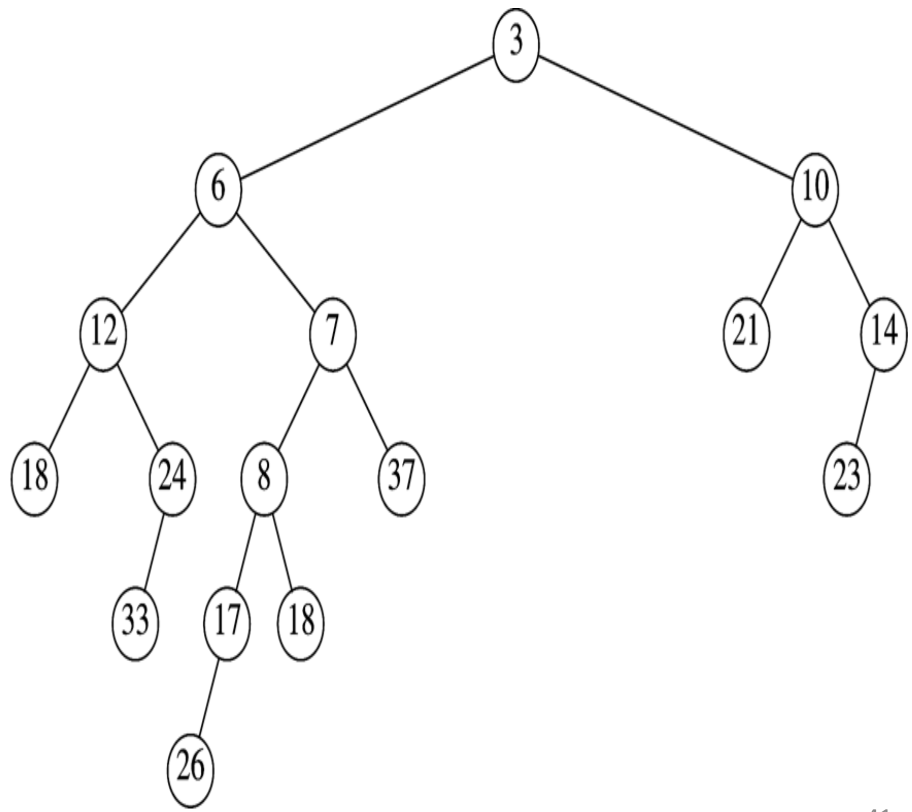
(2) leftist: Merge H2 with H1’s right subheap

Final Result (skew heap)

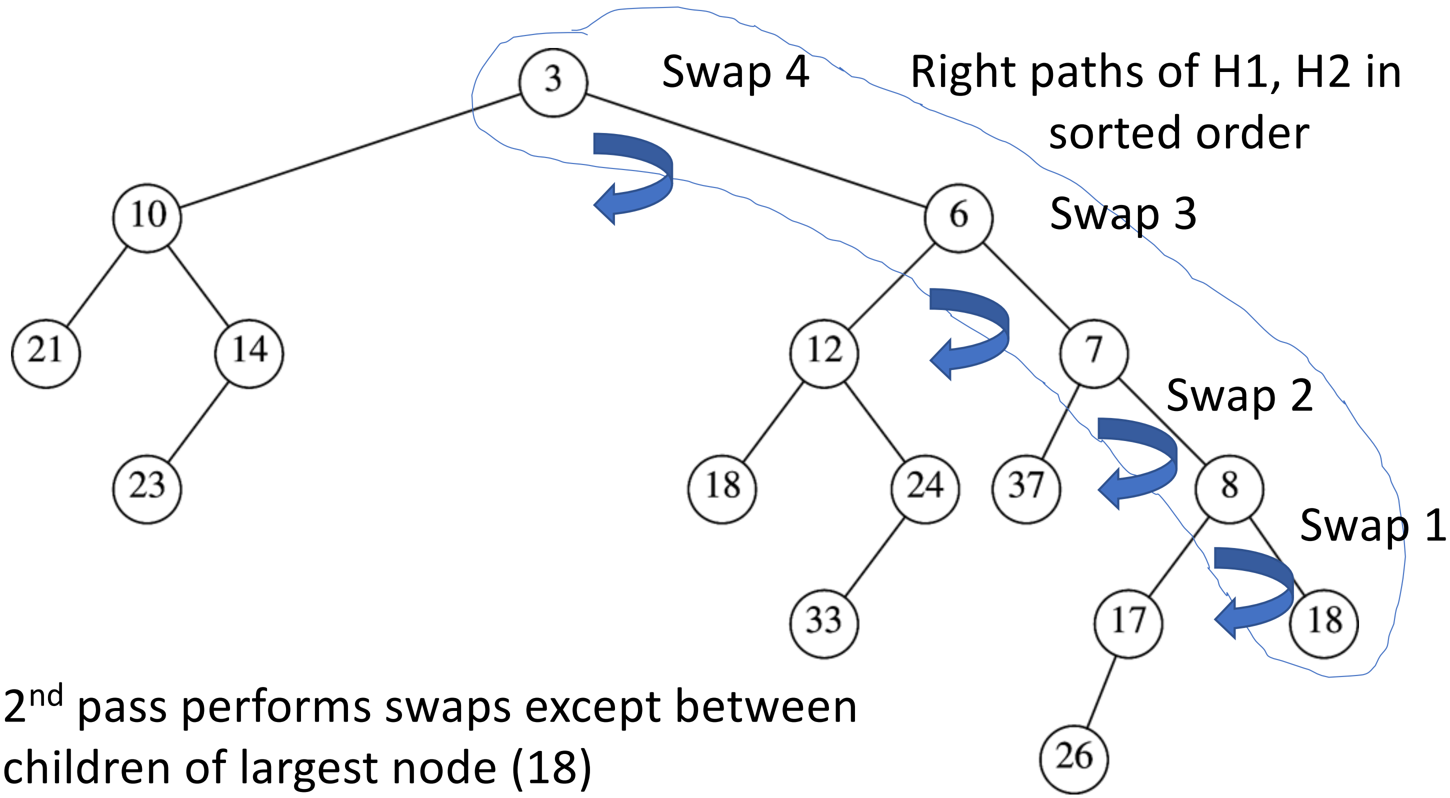


Compare w/ leftist heaps





Non-recursive merge (first pass)



2nd pass performs swaps except between children of largest node (18)

Leftist vs. Skew

- +Skew: no need to store npl()
- +Skew: O(logN) amortized
- +Leftist: O(logN) per operation
- -Skew: individual operations could be long
- -Skew: right path can be long on expensive operations => merge may fail due to stack overflow

Binomial Queues

- O(logN) worst case for Merge, Insert, DeleteMin
- O(1) average case for Insert

Binomial Queue

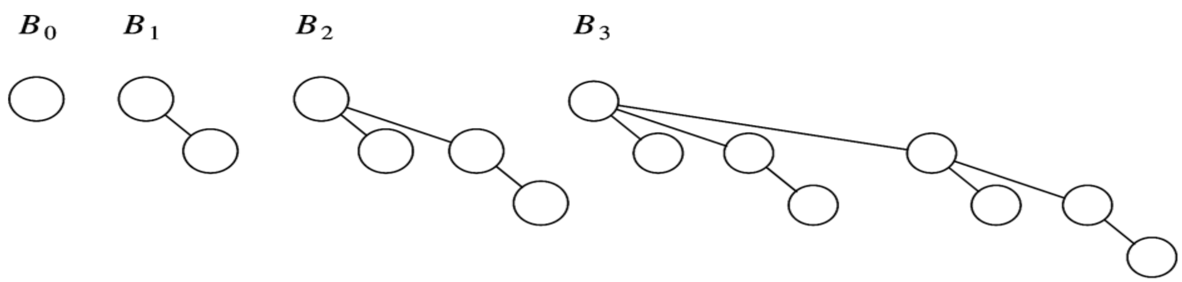
- Collection of heap-order trees (**forest**)
- Each heap-order tree is a **binomial tree**.

Binomial trees

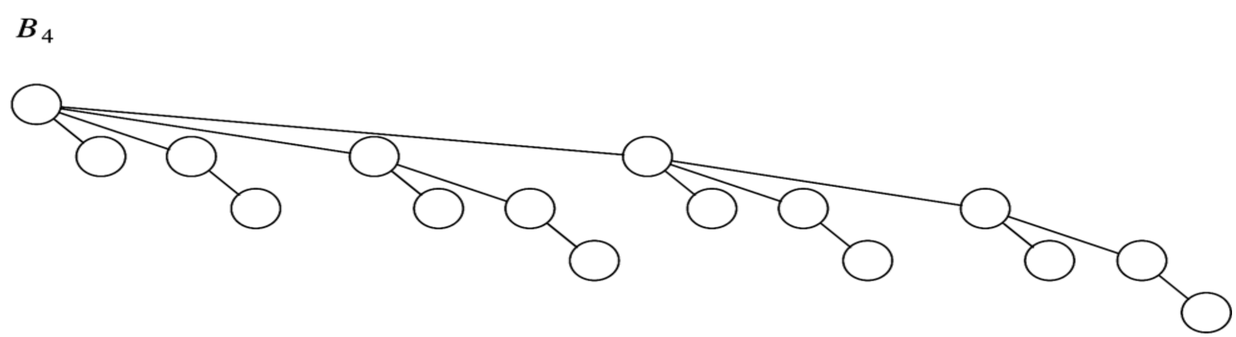
•At most one binomial tree for every height

•Recursive definition:

  •B_k is defined by creating a root, and placing B_0, B_1, B_2, ..., B_{k-1} as children of that root



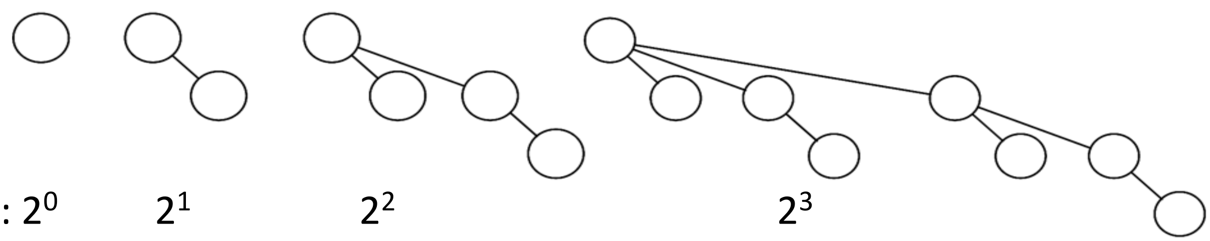
Binomial trees of height 0, 1, 2, 3, and 4



Binomial trees

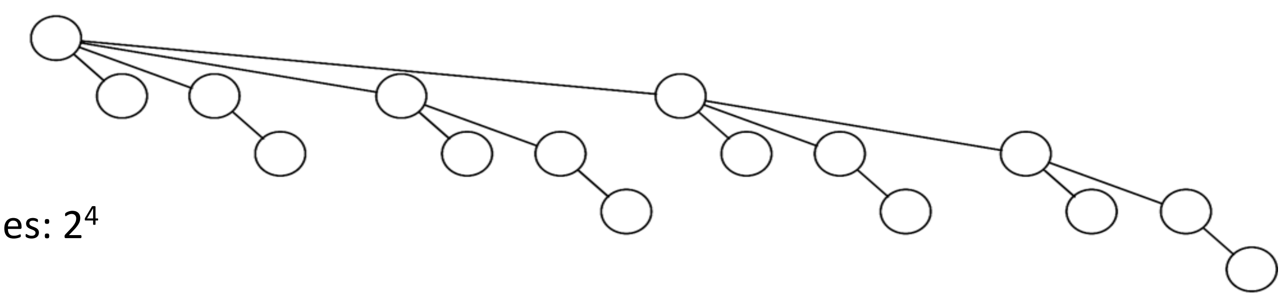
- Bk: height = k
  # of nodes = 2^k

B_0 B_1 B_2 B_3



# of nodes: 2^0 2^1 2^2 2^3

B_4 Binomial trees of height 0, 1, 2, 3, and 4



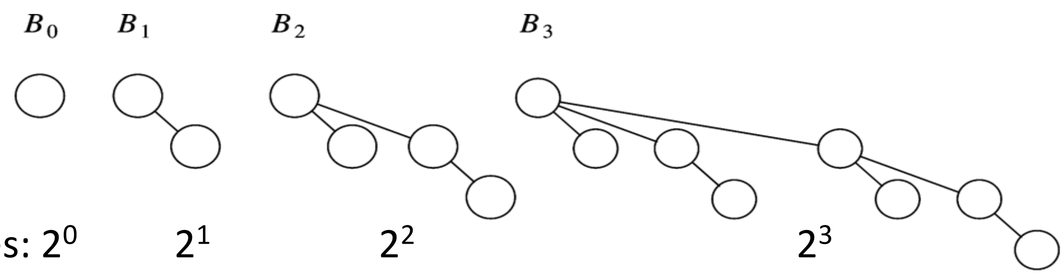
# of nodes: 2^4

Binomial trees

- Bk: height = k

  # of nodes = 2^k

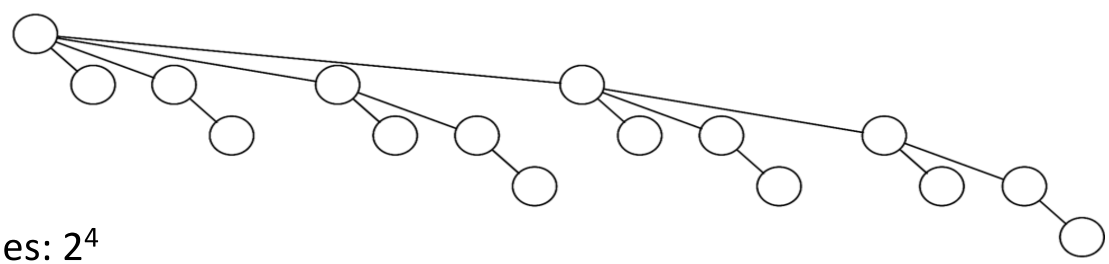
  Number of nodes at depth d is $\binom{k}{d}$



# of nodes: 2^0 2^1 2^2 2^3

B_4

Binomial trees of height 0, 1, 2, 3, and 4



# of nodes: 2^4

Binomial Queue

- Each binomial tree is heap ordered
- Suppose the binomial queue stores N elements
- For example create a b.q. that stores 13 elements
- Use a forest of binomial trees.
- The binary representation of 13 is 1101, since

$$13 = 2^3 + 2^2 + 0 + 2^0$$

- Use b.qs B0, B2, and B3

Binomial Queue

- Each binomial tree is heap ordered
- Suppose the binomial queue stores N elements
- For example create a b.q. that stores 13 elements
- Use a forest of binomial trees.
- The binary representation of 13 is 1101, since

  $13 = 2^3 + 2^2 + 0 + 2^0$
- Use b.qs B0, B2, and B3

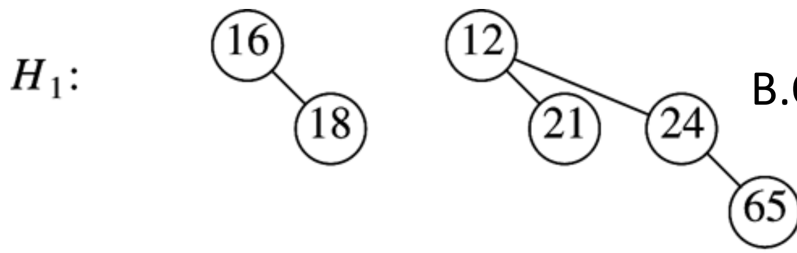
2 | 13
2 | 6 1
2 | 3 0
  | 1 1

1101

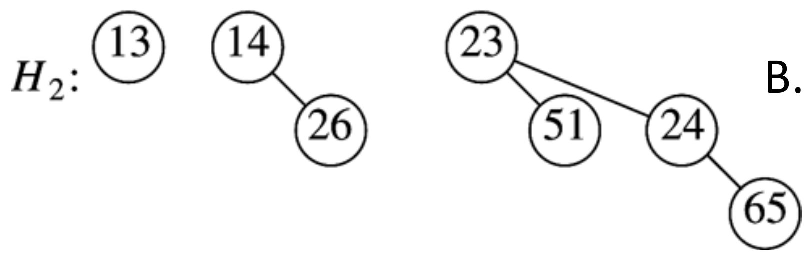
Binomial Queue

- The binary representation of 14 is
- 1110, since
  - $14 = 2^3 + 2^2 + 2^1 + 0$
  - Use b.qs B1, B2, and B3

Binomial queue (example)



B.Q. of 6 elements



B.Q. of 7 elements

B.Q. operations

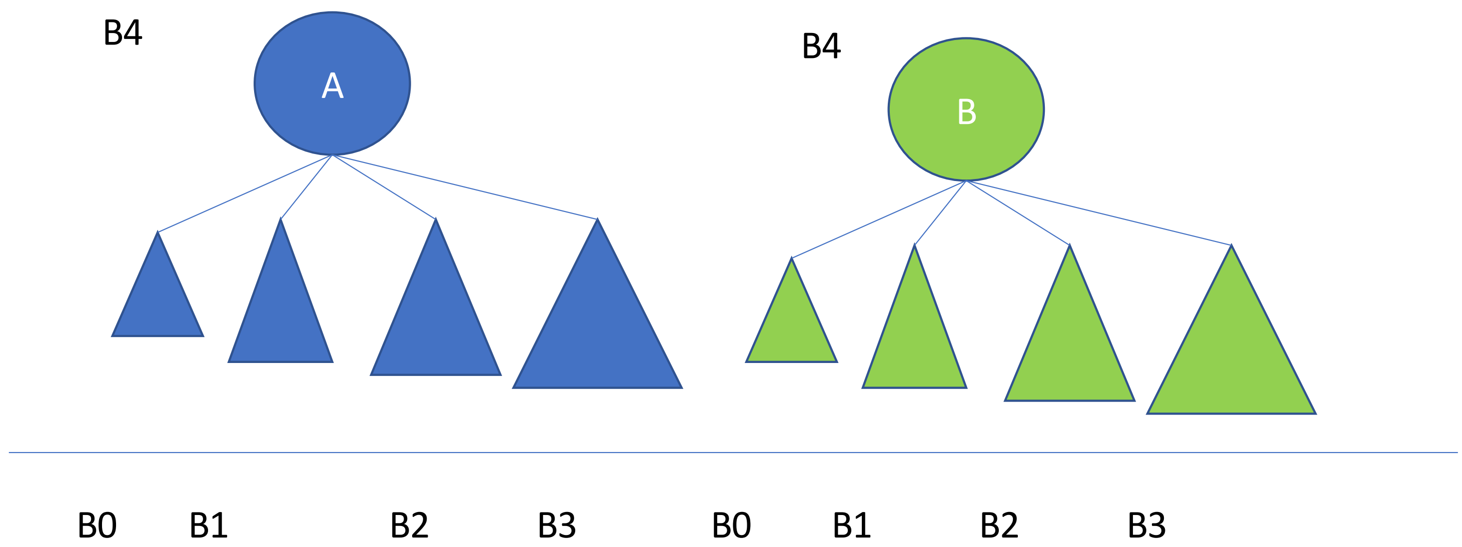
- FindMin():
  - Scan roots of trees.
  - Select the minimum.

- A queue of N elements would have at most how many trees?
  - Therefore, FindMin() is a O(logN) operation

B.Q. operations

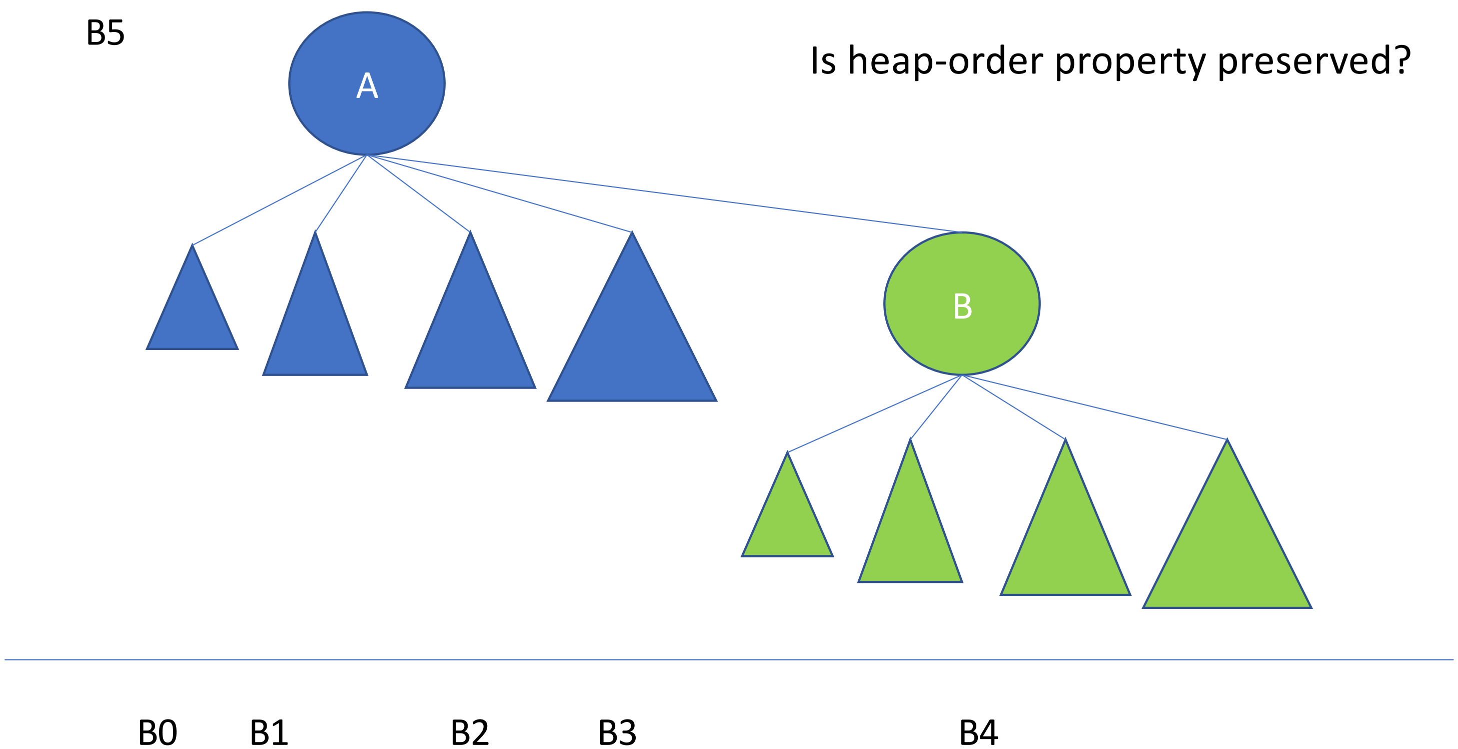
- Merge(H1, H2): ?
  - Merge two BQs ?
  - Merge two Binomial trees?
  - Merge two B0 trees?

Merge two bin. trees of same height (A<B)



Merge two bin. trees of same height (A<B)

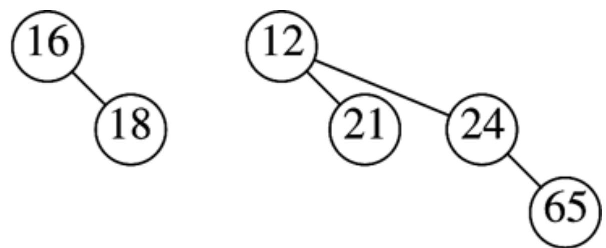
Is heap-order property preserved?



Merge

B0 B1 B2

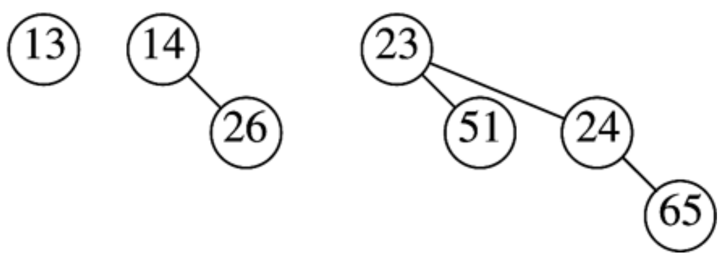
H_1:



B.Q. of 6 elements

B0 B1 B2

H_2:



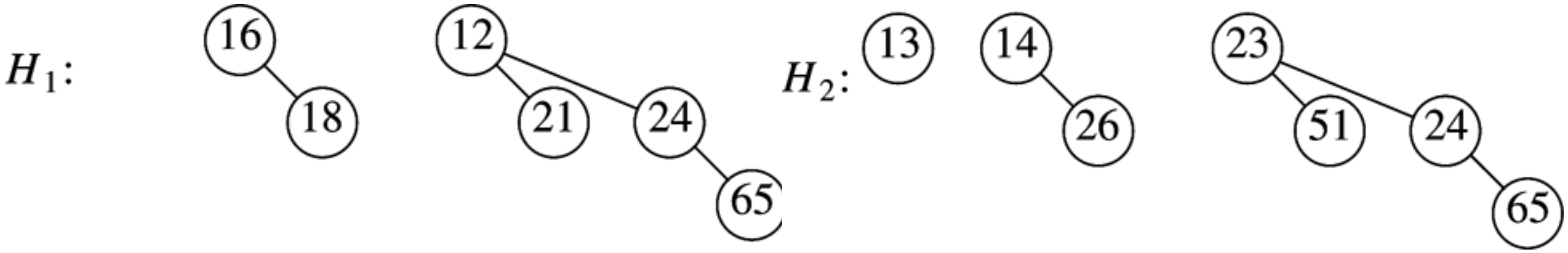
B.Q. of 7 elements

Result:

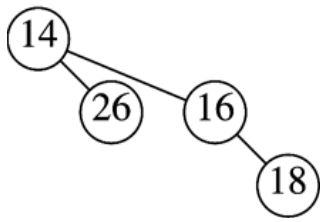
  110 (6)
+ 111 (7)

  1101 (13) elements

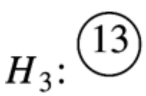
Result



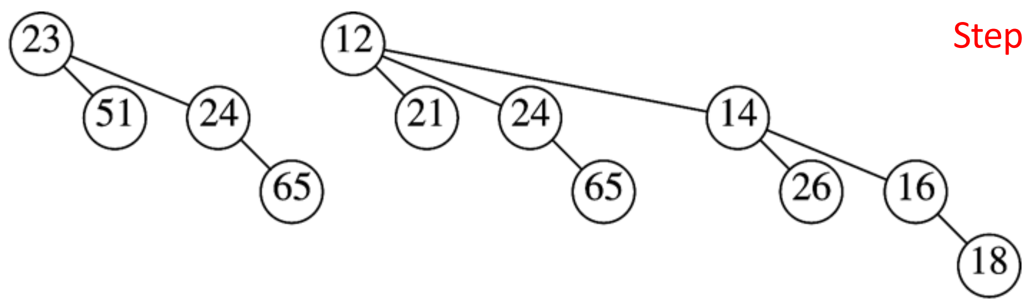
Step 2



B0 B1 B2 B3



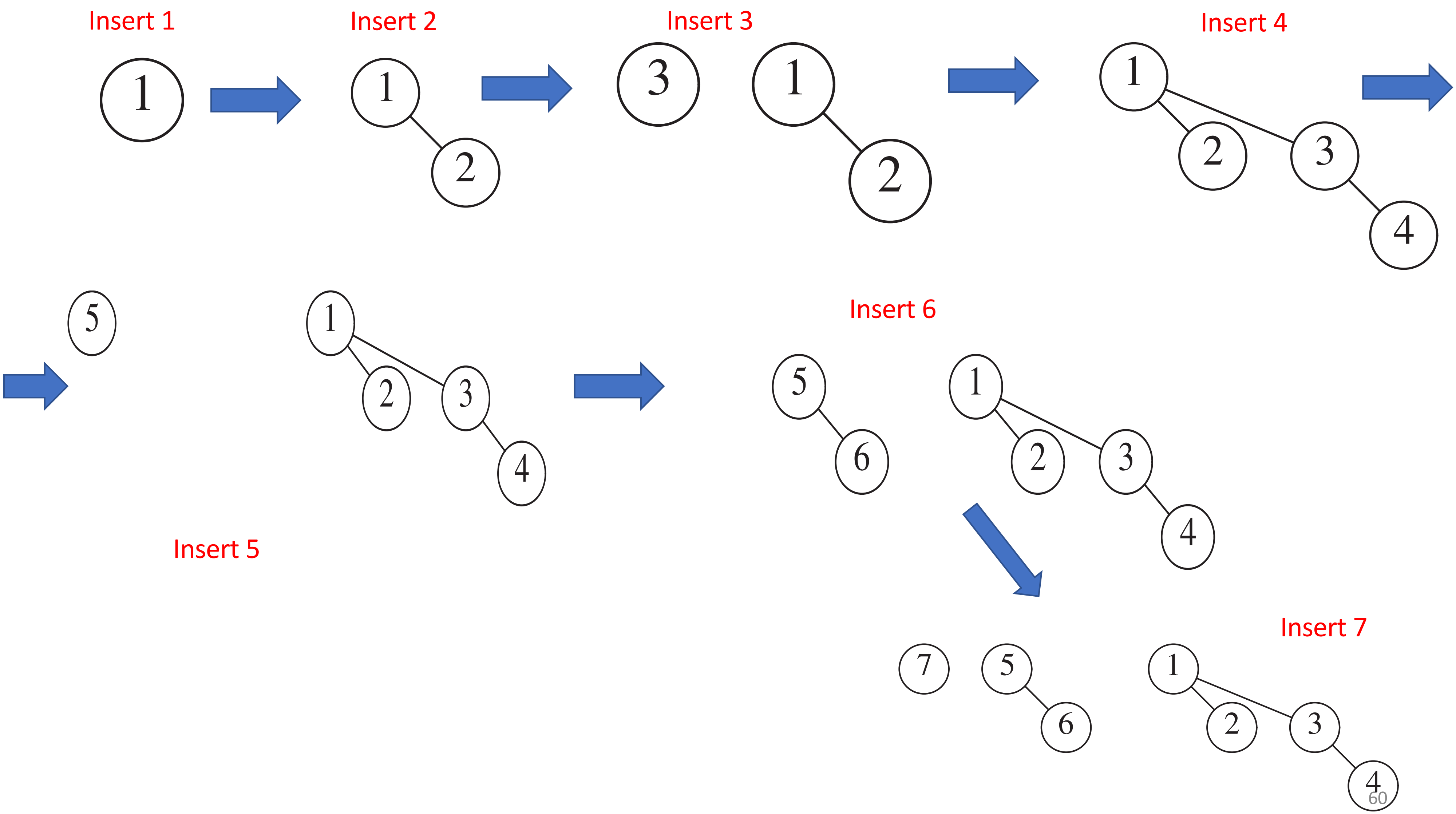
Step 1



Step 3

Merge

- O(logN) worst-case (why?)
- Insert N elements: O(N) worst-case time
- Insert 1 through 7 in an empty binomial queue


Insert 1
1
Insert 2
1
2
Insert 3
3
1
2
Insert 4
1
2
3
4
5
1
2
3
4
Insert 5
Insert 6
5
6
1
2
3
4
Insert 7
7
5
6
1
2
3
4

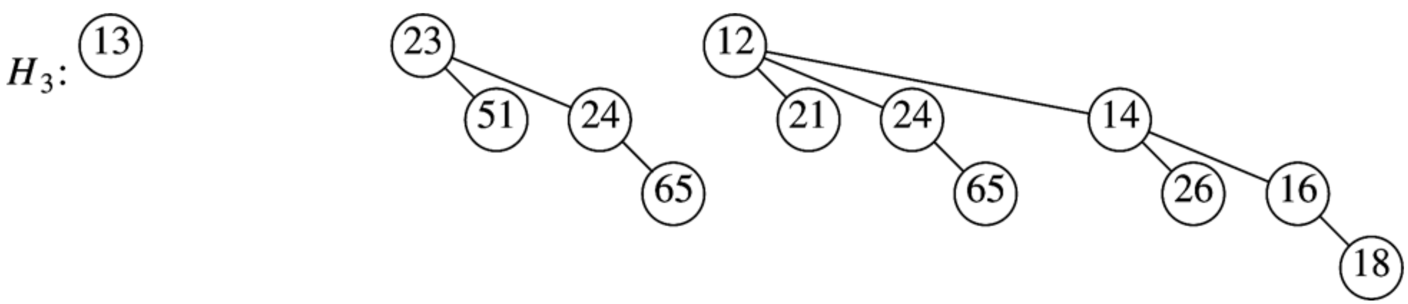
DeleteMin

- H is the original heap
- Find tree containing min root (suppose it is Bk)
- Delete tree from H => new queue H’
- In Bk, delete its root

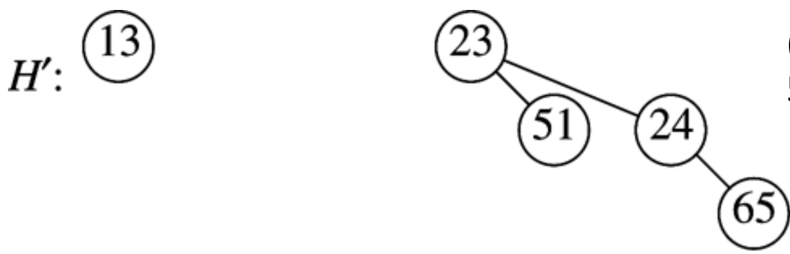
  => forest of trees under Bk (B0, …, Bk-1): this is a heap H’’
- Final step: Merge H’ with H’’

Example (DeleteMin)

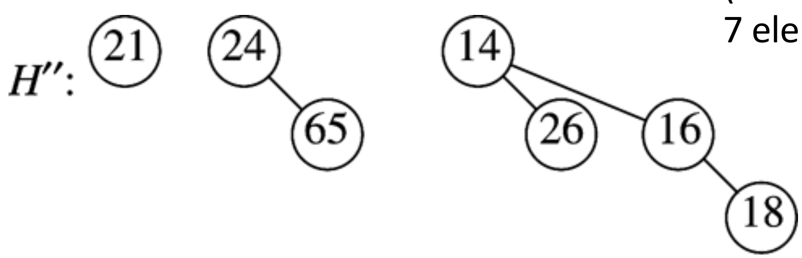
B0 B1 B2 **B3**



Example (DeleteMin)



(after deleting B3)
5 elements

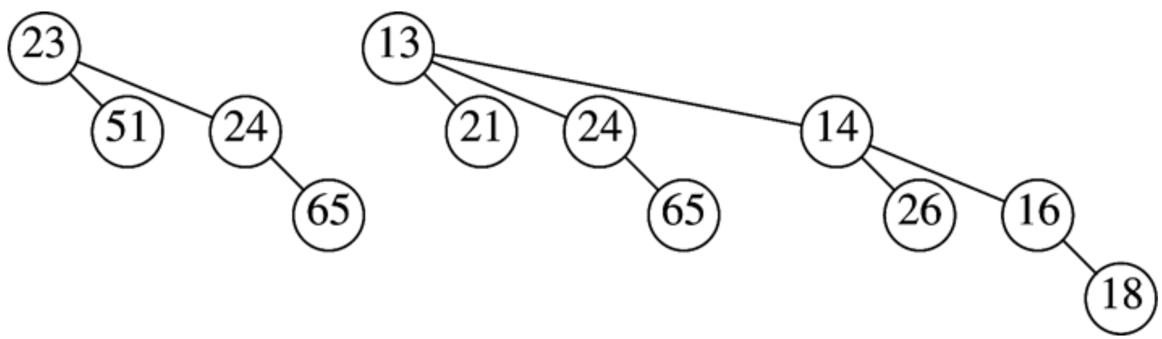


(after deleting B3’s root)
7 elements

Example(DeleteMin)

Merge H' , H''

```
    101   (5)
 +  111   (7)
 -------
    1100  (12) elements
```

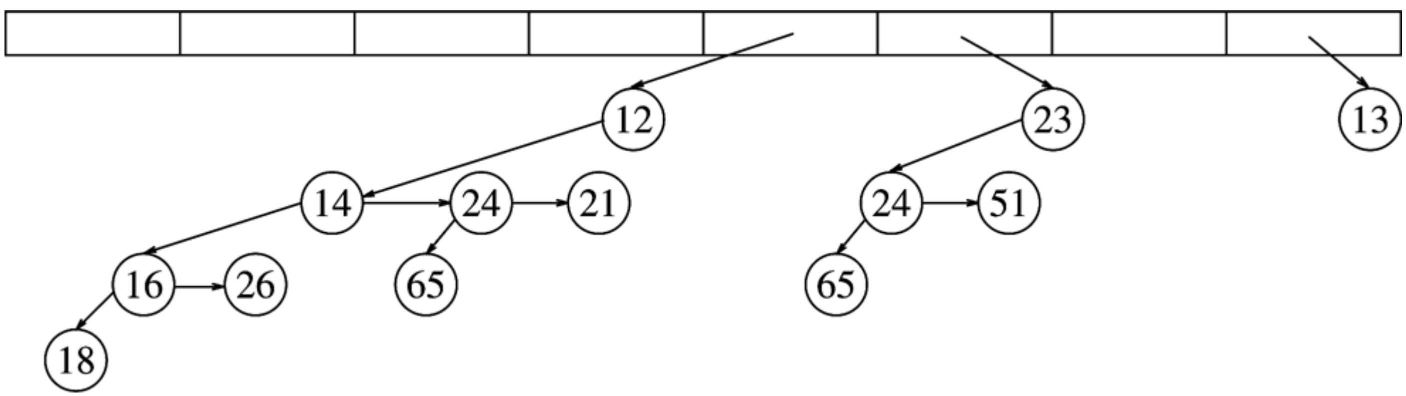


DeleteMin

- H is the original heap
- Find tree containing min root (suppose it is Bk) [O(logN)]
- Delete tree from H => new queue H’ [O(logN)]
- In Bk, delete its root [O(logN)]

  => forest of trees under Bk (B0, …, Bk-1): this is a heap H’’
- Final step: Merge H’ with H’’ [O(logN)]

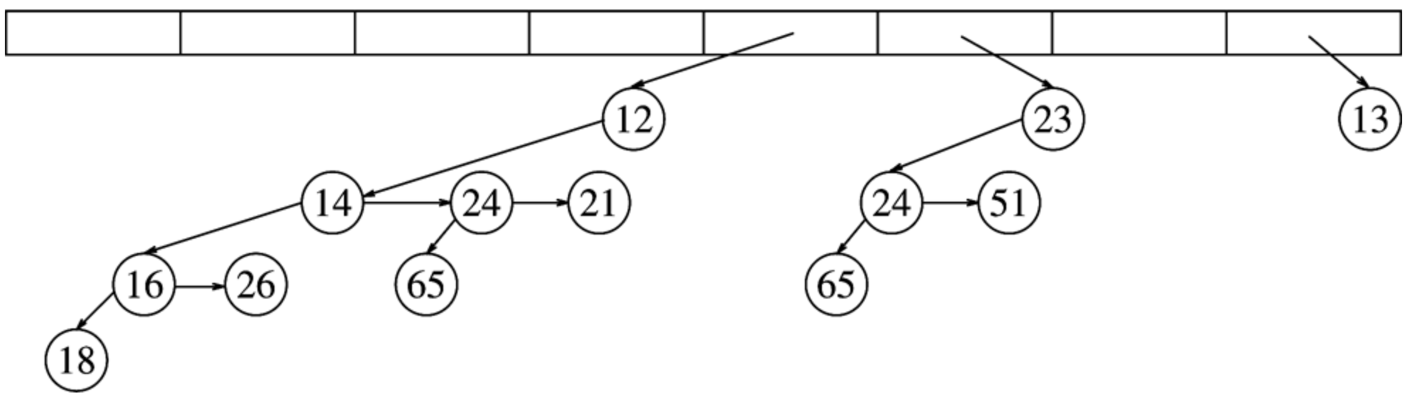
Representation



Representation

- Maintain a list of pointers to the roots of the binomial trees
- Order the trees by height to make merge efficient

```
vector<BinomialNode *> the_trees_;  // An array of tree roots.
int current_size_;                  // Number of items in the queue.
```



Summary

- Basic heap
- Heaps with efficient merge operation
- Leftist: power of recursion
- Skew: self adjusting
- Binomial: simple but powerful idea

Summary: complexity

- Basic binary heap
  - $O(\log n)$ for insert and deleteMin
  - $O(1)$ average for insert
  - $O(n)$ to build the heap
  - STL priority_queue probably works like this
- Heaps with $O(\log n)$ merge operation
  - Leftist – Simple recursive data structure
    - Does not have $O(1)$ average for insert.
  - Skew – Amortized version of Leftist heap
  - Binomial – Based on a forest. Simple to describe and has the same complexity guarantees as the binary heap.