TOPIC 2A
DATA STRUCTURES: HEAPS

DATA STRUCTURES

Data structures are used for organizing data in memory.

Algorithms and Data structure go hand in hand

- Without appropriate data structures algorithms would be slow
- Without algorithms data cannot be manipulated

Data Structures defined by their operations

- For dynamic sets: Insert, Delete, Search, Minimum, Maximum, Predecessor (in a sorted list), Successor (in a sorted list)
- More complicated operations for complex data types

Abstract Data Type (ADT)

- Operations that can be performed on the data structure and the complexities of the operations
- Example: Heaps

Data Structure

- The implementation of the ADT, depends on the language, the platform, etc.
- Example: Heaps implemented as arrays

DESCRIBING A DATA STRUCTURE

API (Application Programming Interface):

- Functions provided by the data structure and their complexities

Invariants

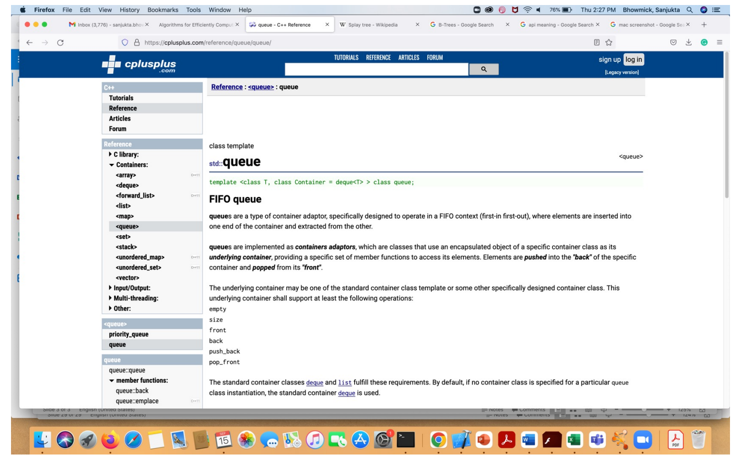
- Properties that hold true during the lifetime of the data structure

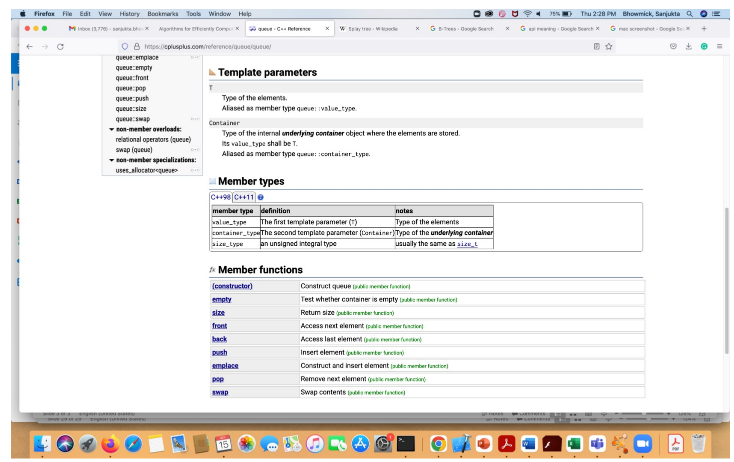
Data Model

- How is the data stored; list, array, etc.

Algorithms

- Algorithms to support the API functions





PRIORITY QUEUES

Given a dynamic data set, enable access to element with highest priority

- Queue: First in First out (Priority: highest time stamp)
- Stack: Last in First out (Priority: lowest time stamp)
- Used in scheduling, shortest paths, efficient routing, etc.

More general: heaps

- Max heap: Element with highest value
- Min heap: Element with lowest value

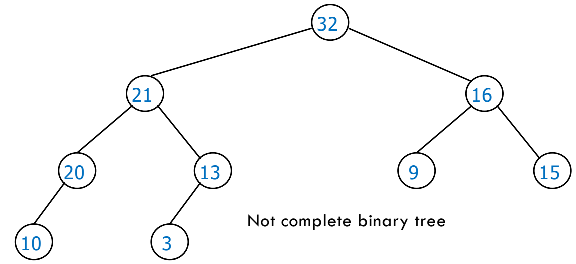
Operations

- Top: remove element with highest priority from top
- Peek: check what is the element with the highest priority
- Insert: insert an element, along with its priority
- Delete: remove an element
- Update: update the priority of an element

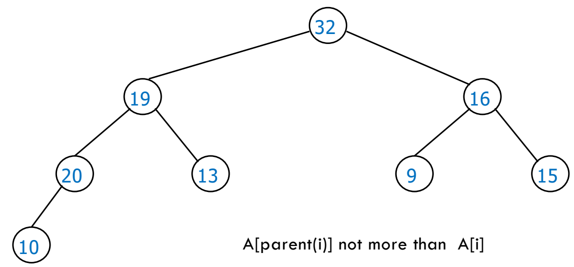
BINARY HEAP

Invariant Properties

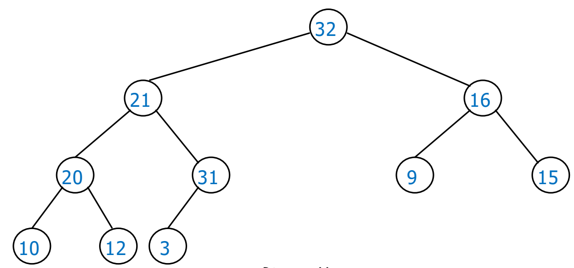
- Is a complete binary tree
  - Every level except possibly the last is filled and the leaves are as far left as possible (fill from left to right)
  - What is the benefit ?
- For a maxheap: every node i other than the root, A[parent(i)] > A[i]
  - Parents are greater than children
  - Reverse if minheap
  - What does that imply about the root ?



Not complete binary tree



A[parent(i)] not more than A[i]



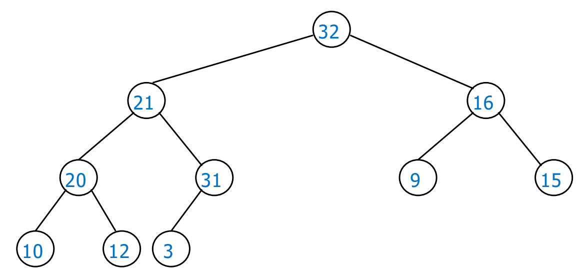
Binary Heap

BINARY HEAP

For any element in array position i, the left child is in position $2i$, the right child is in the cell after the left child $(2i+1)$, and the parent is in position $\lfloor i/2 \rfloor$

The operations required to traverse the tree are extremely simple and very fast on most computers.

32	21	16	20	31	9	15	10	12	3
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*



UPDATING—INCREASE KEY

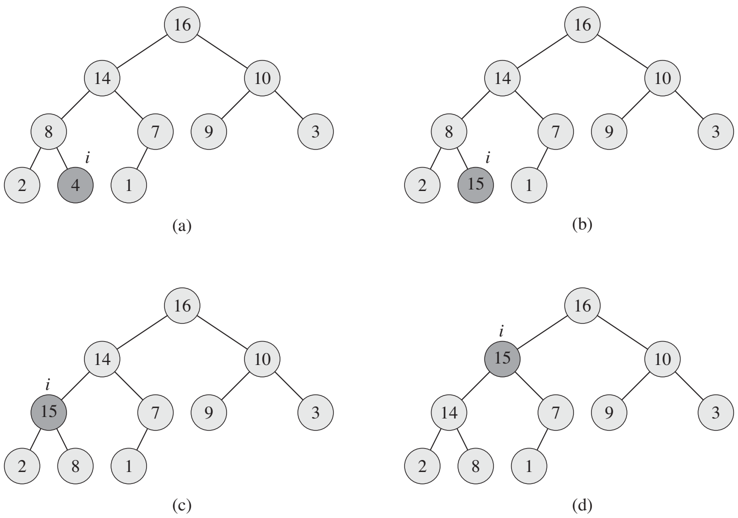


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

Complexity: O(logn)

BUILD HEAP

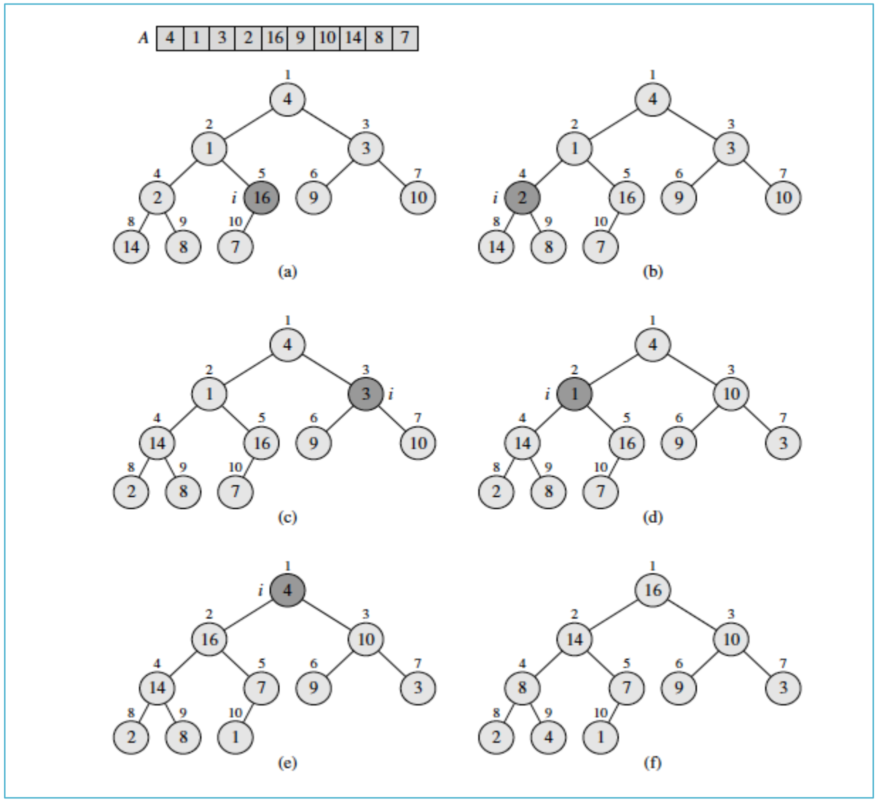
Fill the heap in order the numbers appear in the array A

For i=length[A]/2 to 1

- Start with the rightmost nonleaf node to root
- Percolate down until heap property is achieved

  A[parent[i]]>A[i]

Max Heap



COMPLEXITY OF BUILDING HEAP

To build the heap, we have to heapify n elements

Each element takes O(log n) time = height of the tree

What is the total time to build the heap

- n*O(logn)=O(nlogn)
- However, by creating a max heap we can only know the element with the maximum value
- We could have had the information in O(n) time by just going through the array
- Why is build heap more expensive ?

TIGHTER BUILD-HEAP TIME

Each node does not traverse any more than its height.

There are at most ceil($n/2^{h+1}$) nodes at height h.

Therefore total number of traversals

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

Tighter Build-Heap Time

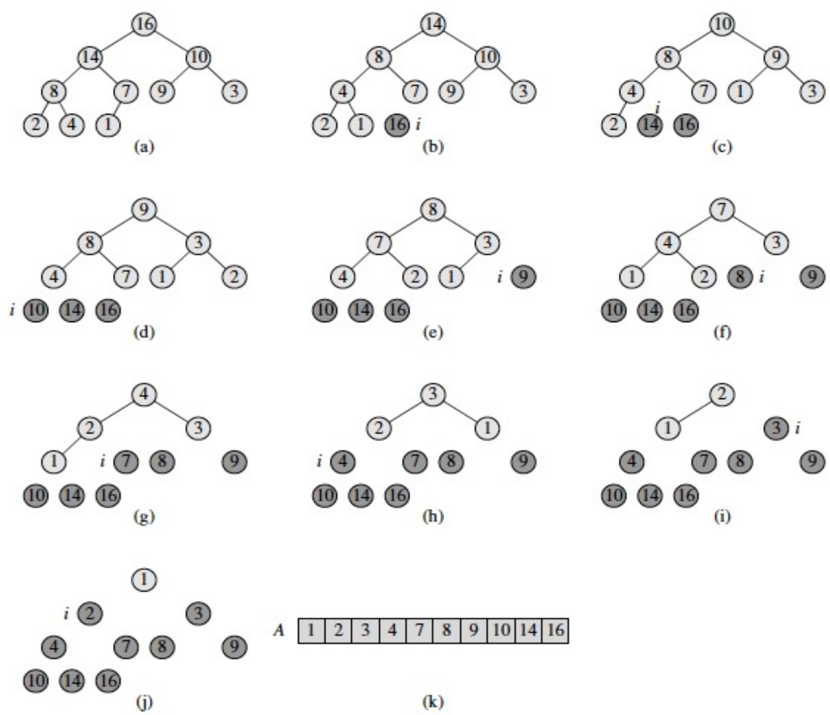
Geometric Series $\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$.

If x<1 and n tends to inf $\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$.

Taking the derivative on each side $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1 - x)^2}$

Set x=1/2 to compute bound on build heap to be O(n)

HEAPSORT




```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] − 1
        MAX-HEAPIFY(A, 1)
```

Replace element to be deleted by the last in array.
Delete the last element
Update heap to maintain heap property

Deleting elements takes O(logn)
Heapsort=Deleting n elements
O(log(n))+ log(n-1) +… 1) =O(logn!)=O(nlogn)

COMPLEXITIES OF HEAP OPERATIONS

Top : O(1)

Peak: O(1)

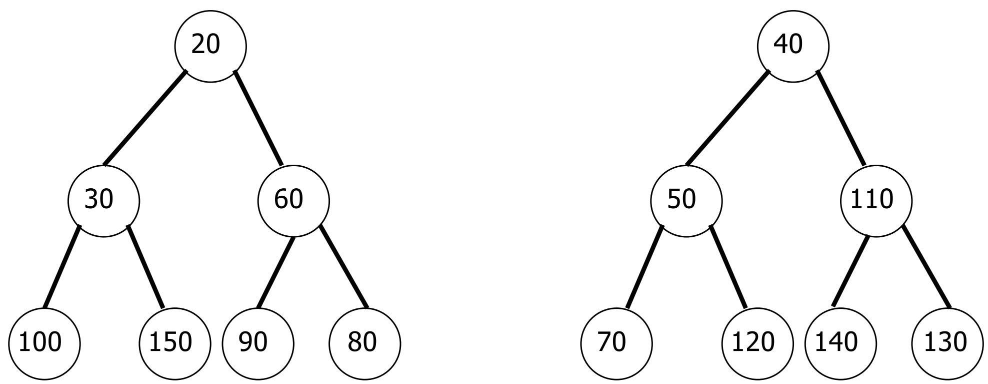
Insert: O(logh), worst case O(logn)

Delete: O(logh), worst case O(logn)

Update: O(log h), worst case O(logn)

- How do we combine two heaps ?
- Can we do Insert and Update in O(1) ?

MERGING TWO HEAPS



If we concatenate the arrays and recreate the heap O(m+n)
If smaller heap has n elements, and we just insert these elements, then total cost is O(n)
Can we do better ?

LEFTIST HEAP

A leftist heap is a binary heap where

- Element of highest priority (max/min) is at top
- Left(S_value)>= Right(S_value)

- S_value=distance to leaf
- S_value of leaf =0
- S_value of node with one child=0

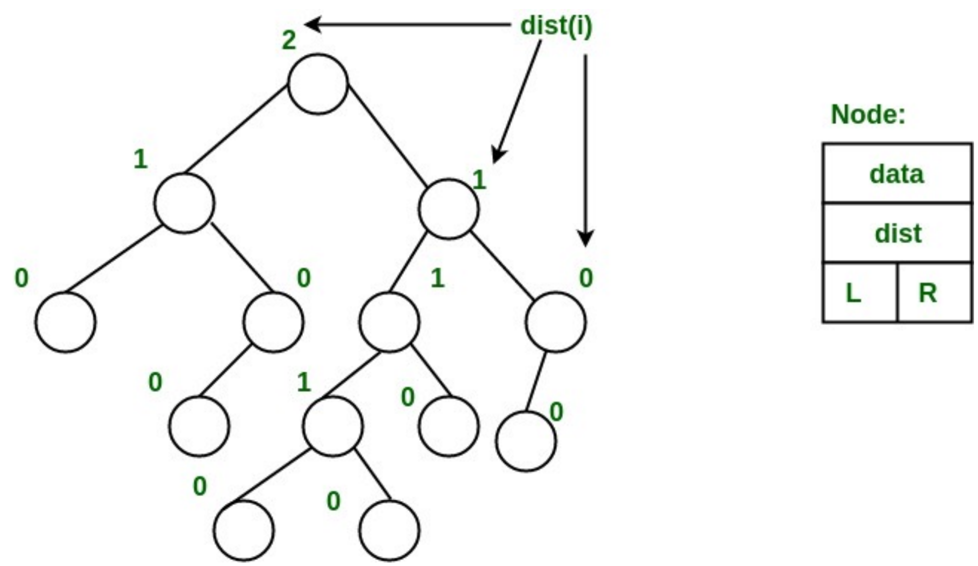


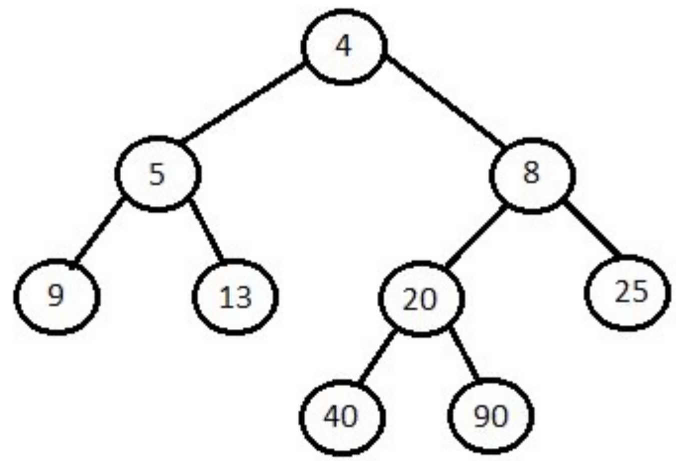
Image from https://www.geeksforgeeks.org/leftist-tree-leftist-heap/

BUILDING LEFTIST HEAP

Insert new node as rightmost leaf

Check if S_value criteria is maintained

Otherwise swap the left and right subtrees to maintain the criteria



https://www.cs.usfca.edu/~galles/visualization/LeftistHeap.html

Image from https://iq.opengenus.org/leftist-heap/

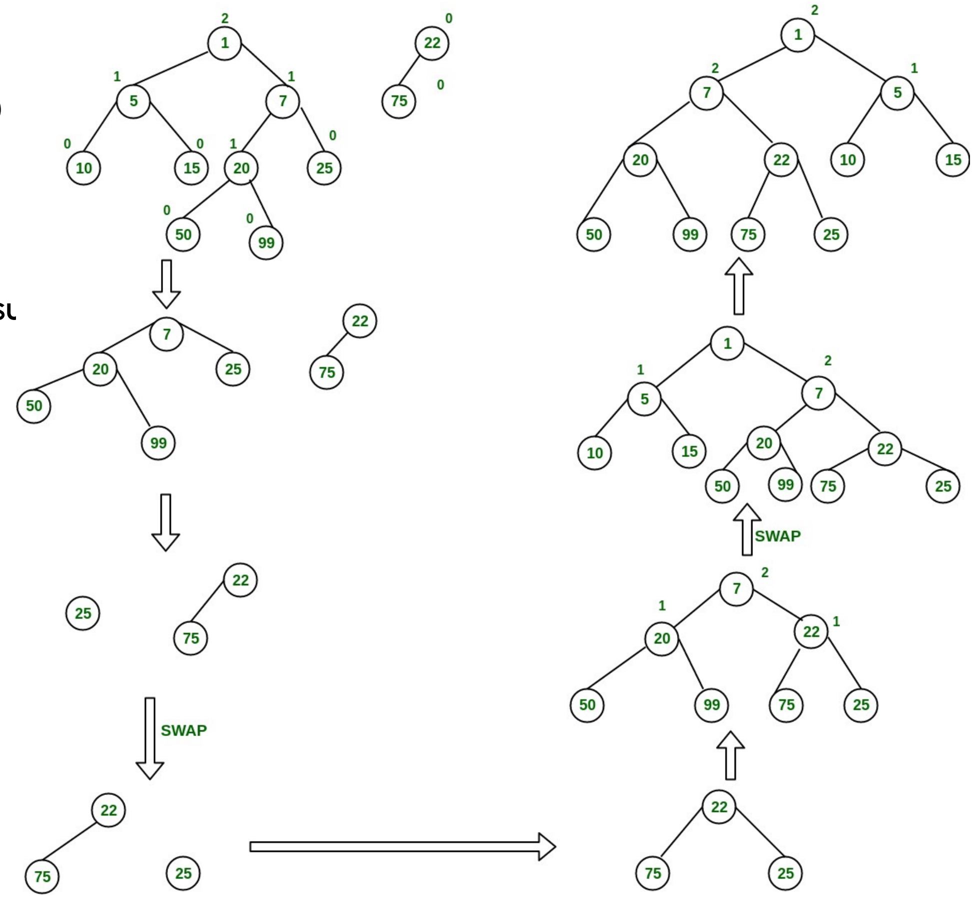
MERGING LEFTIST HEAPS

Merge(H1, H2)

- If H1 or H2 contains one node, insert as usu
- If root(H1)<root(H2)
  - right_subtree(H1)=new_H2
  - H2=new_H1
- Else
  - right_subtree(H2)=new_H2
  - H1=new_H1

  Swap to maintain S_value criteria

  as needed
- Merge(new_H1, new_H2)



https://www.geeksforgeeks.org/leftist-tree-leftist-heap/

DELETION

Remove Root

This will create two trees

Merge them

Complexity of Insertion (one element), Deletion (one element) and Merging (smallest tree has n elements) are all O(logn)

Why do we get O(logn), given that the tree is not balanced ?

PROPERTY OF LEFTIST HEAP

Given a heap with n elements, the shortest path from root to a leaf is O(logn)

- Since Right(S_value)<=Left(S_value); therefore one of the shortest path will to the rightmost leaf
- Let the Right(S_value) of root be x
- That means there are at least x nodes from root to leaf.
- Since nodes without two children have S_value 0, therefore all nodes with S_value >1 in the path will have two children.
- Thus there are at least 2^x-1 nodes
- 2^x-1 <=n (n is total number of nodes)
- x<=O(log(n))

Since we are prioritizing new insertions in the right subtree, therefore complexity if O(logn)

BINOMIAL QUEUES

Binomial queues support all three operations (insertion, deleteMin, and merge) in O(log N) worst-case time per operation, but insertions take constant time on average.

BINOMIAL QUEUE STRUCTURE

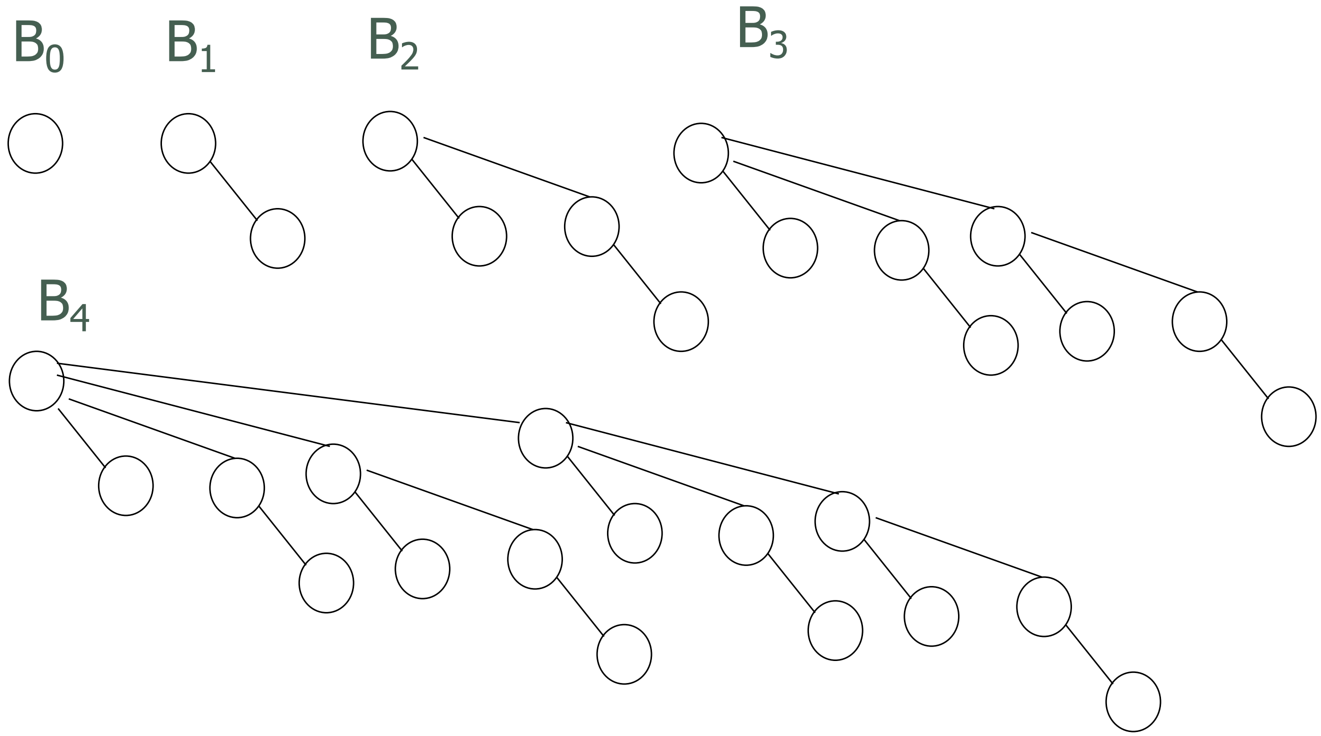
Binomial queue is not a heap-ordered tree but rather a collection of heap-ordered trees, known as a forest.

Each of the heap-ordered trees is of a constrained form known as a binomial tree.

HOW TO CONSTRUCT A BINOMIAL TREE?

1. A binomial tree B_0 of height 0 is a one-node tree;
2. 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}.

BINOMIAL TREES B_0, B_1, B_2, ...



PROPERTIES

A binomial tree, B_k, consists of a root with children B_0, B_1, …, B_{k-1}.

The height of a binomial tree, B_k, is k.

Binomial trees of height k have exactly 2^k nodes.

The number of nodes at depth d is the binomial coefficient (root has depth 0 and height k)

- $= k! / ((k-d)! \cdot d!)$.

PRIORITY QUEUE USING BINOMIAL TREES

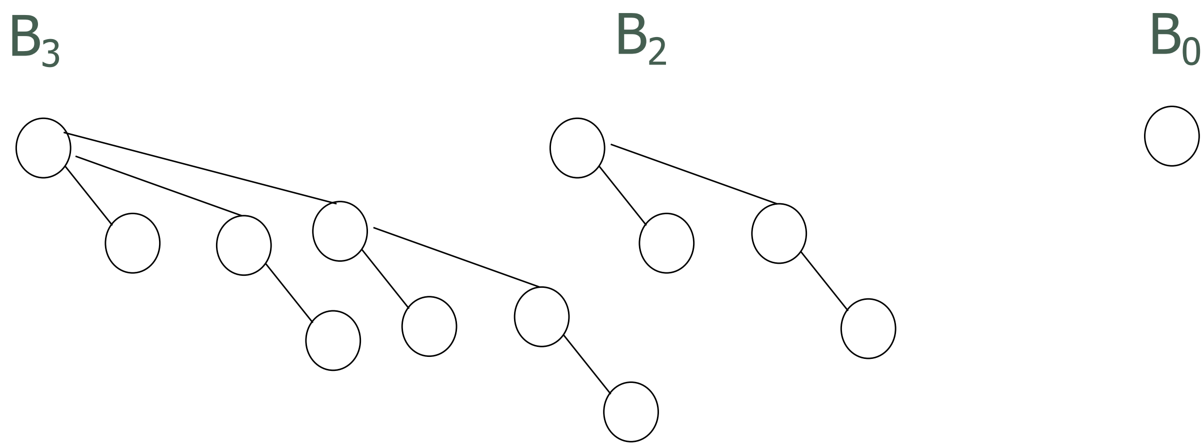
Impose heap order on the binomial trees

Allow at most one binomial tree of any height

Then, we can uniquely represent a priority queue of any size by a collection of binomial trees.

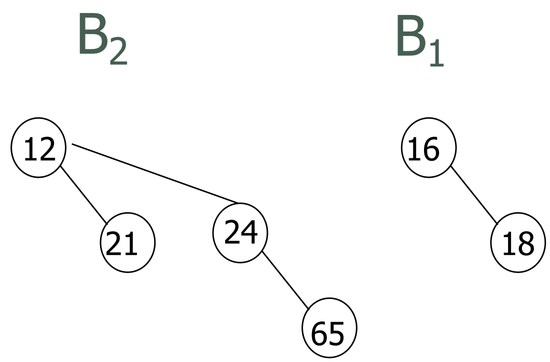
HOW ?

A PRIORITY QUEUE OF SIZE 13



13 = 1101

A PRIORITY QUEUE OF SIZE 6



6 = 110

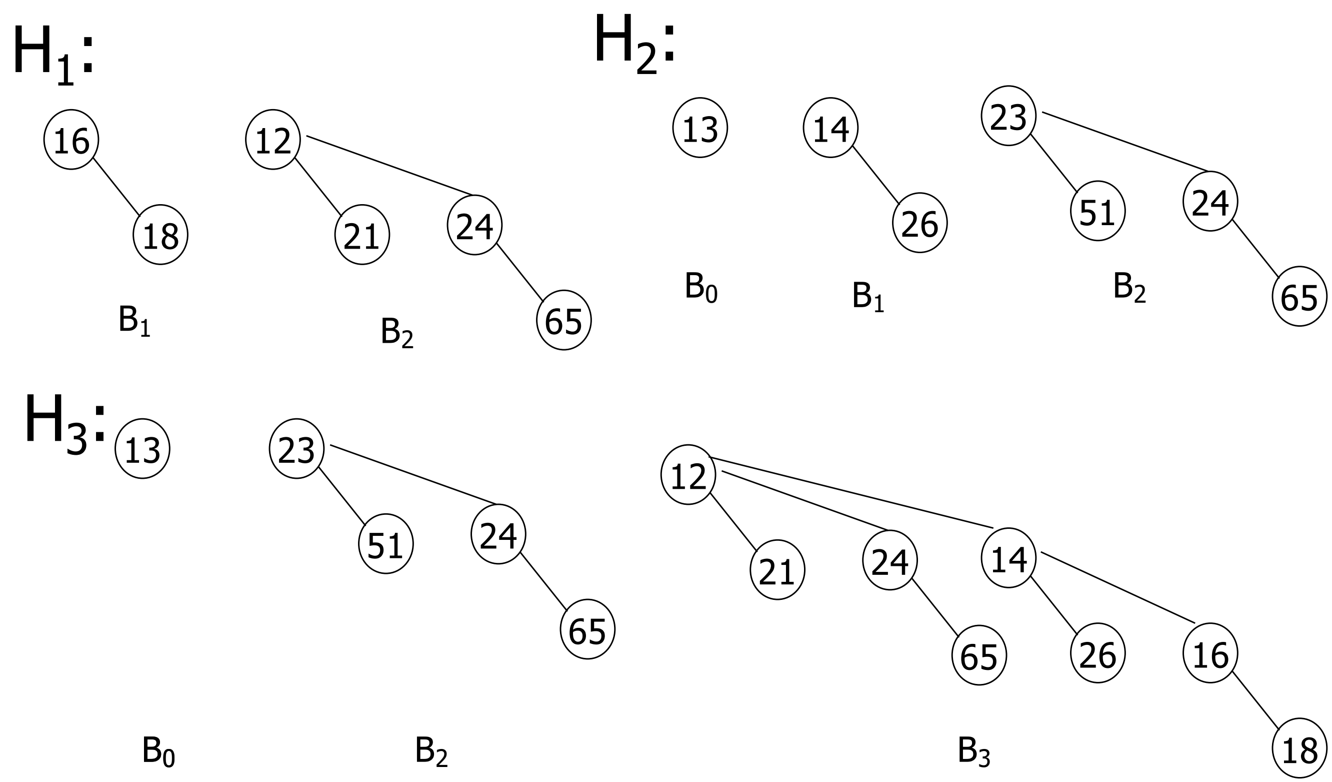
BINOMIAL QUEUE OPERATIONS

Find the minimum element: O(log N)

- The minimum element can be found by scanning the roots of all the trees. Since there are at most log N different trees, the minimum can be found in O(log N) time.

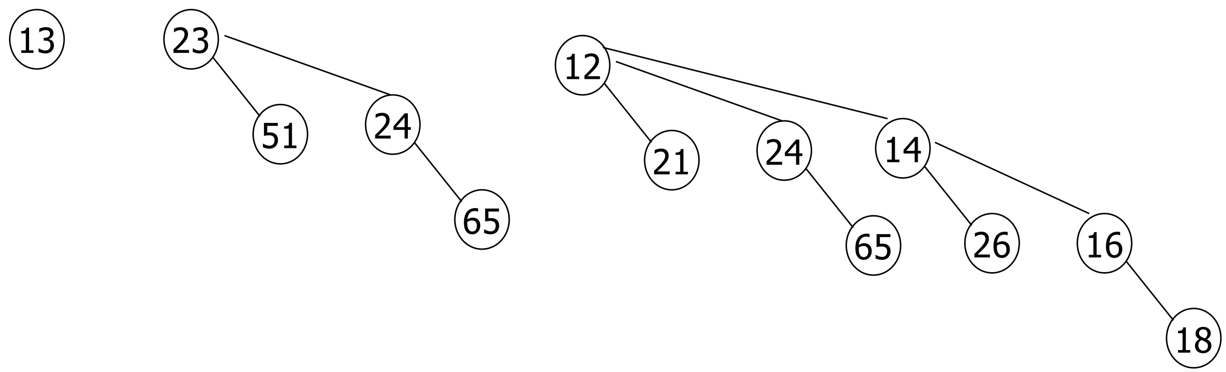
We can maintain knowledge of the minimum and perform the operation in O(1) time.

MERGING TWO BINOMIAL QUEUES

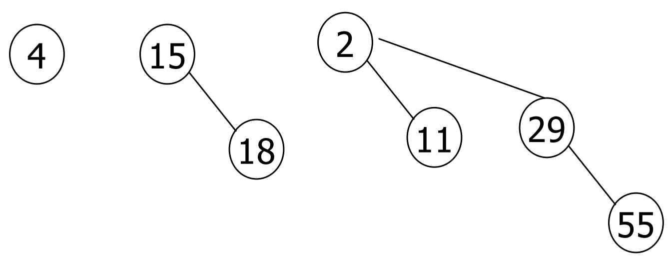


EXERCISE: MERGING TWO BINOMIAL QUEUES

H_1:



H_2:



COMPLEXITY ON MERGE

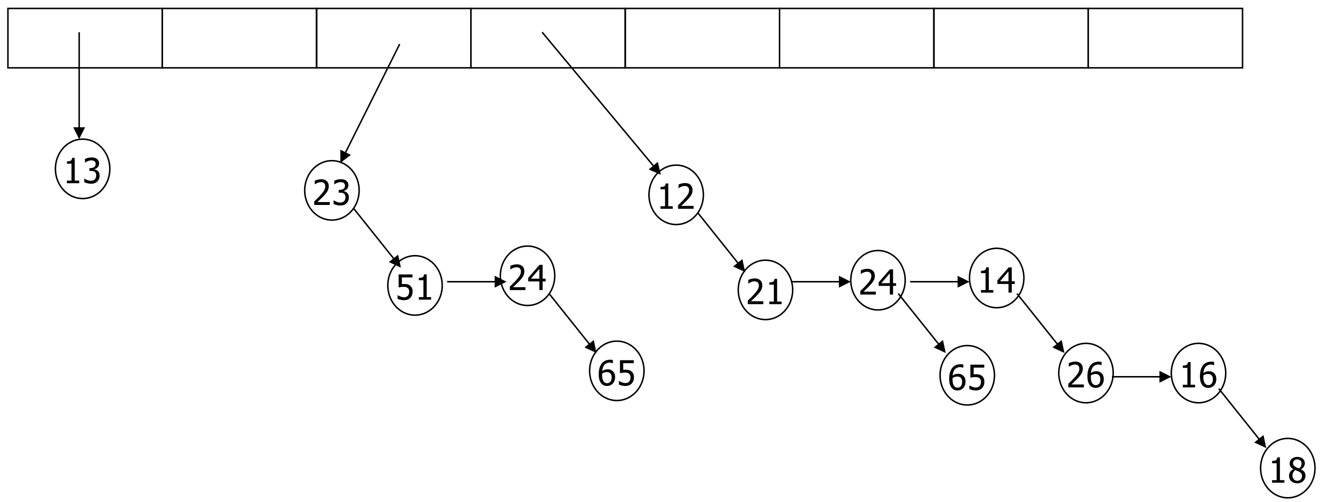
Merging two binomial trees takes constant time.

There are O(log N) binomial trees.

So merge two binomial queues take O(log N) time in the worst case.

To make this operation efficient, we need to keep the trees in the binomial queue sorted by height.

REPRESENTATION OF BINOMIAL QUEUE



INSERTION

Insertion is just a special case of merging: create a one-node tree and perform a merge.

Worst case: O(log N)

Averagely: O(1)

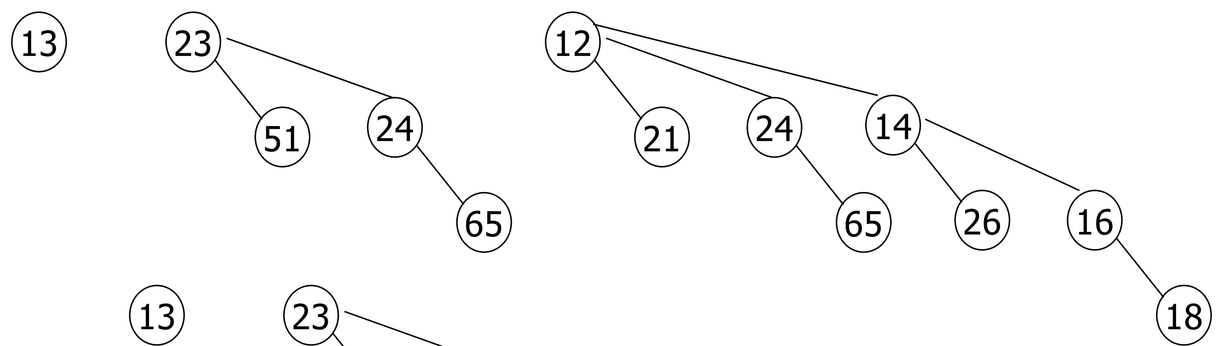
Performing N inserts on an initially empty binomial queue will take O(N) worst-case time

DELETEMIN

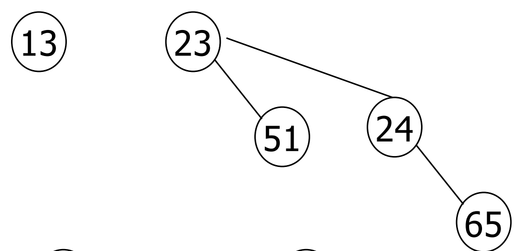
1. Find the binomial tree with the smallest root. Let this tree be B_k, and let the original binomial queue be H.
2. Remove the binomial tree B_k from H, and form a new binomial queue H'.
3. Remove the root of B_k, create binomial trees B_0, B_1, …, B_{k-1}, and form another new binomial queue H''.
4. Merge H' and H''.

DELETEMIN OPERATION

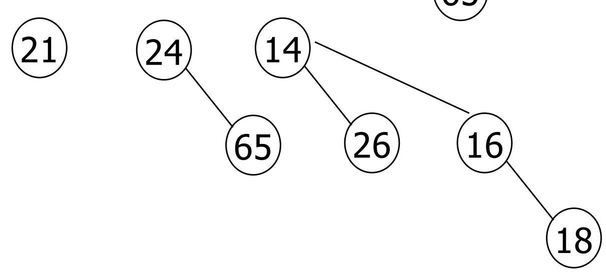
H_3:



H′:



H″:



Merge H′ and H″.

COMPLEXITY ON DELETEMIN

It takes O(log N) time to find the tree containing the minimum element.

Constant time to create the queues H’ and H’’.

Merging these two queues takes O(log N) time.

Entirely O(log N) time

SUMMARY

Binary Heaps: Find minimum is constant; Insertion, Delete in O(logN); Merge is O(n+m)

Leftist heaps: Merge is O(logn). Not balanced

Binomial Heap: Insertion is constant on average. A forest of multiple binomial trees

Other types of heaps:

D-way heaps, Fibonacci heaps, etc.