Data Structures

R. K. Ghosh

IIT Kanpur

Priority Queues & Heaps

Priority Queues & Heaps

Priority Queues

Stores elements using a partial ordering based on priority such that the element of the highest priority is at the head queue.

Heaps

Heaps are data structures for implementing priority queues. Heaps support following two basic operations:

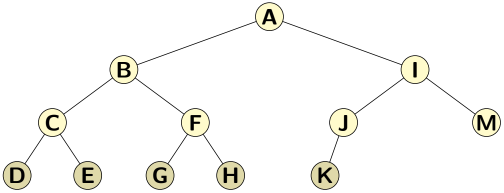
1. **insert()**: Is equivalent to enqueue. Enqueues elements according to their priorities maintaining the queue order.
2. **deleteMin()**: Is equivalent to dequeue. Finds minimum element (of the highest priority) and deletes it from the queue. The priority order of queue is restored by placing the next smallest at the head.

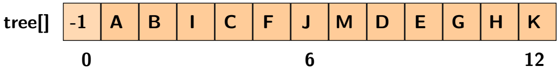
Implementing Priority Queues

- Using unsorted linked lists
  - O(1) **insert**: always insert at the beginning
  - O(n) **deleteMin**: traverse list to find the minimum.
- Using sorted linked list
  - O(n) **insert**: inserts at the correct postion in the list.
  - O(1) **deleteMin**: gets the first element of the list.
- Using sorted array
  - O($n + \log n$) **insert**: inserts at the correct postion in the array then readjusts the array.
  - O(n) **deleteMin**: deletes the minimum, then r eadjusts the array.

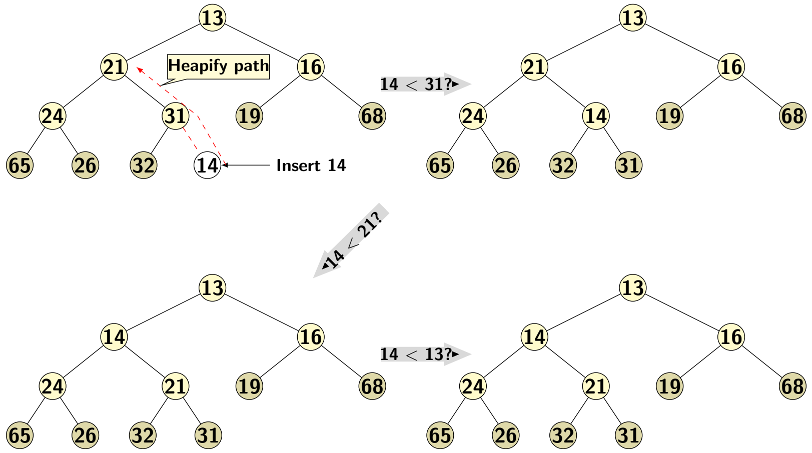
Binary Heap

- The elements are organized in the form of a complete binary tree.
- For storing a complete binary tree use an array.
- The parent of a node at i at stored at $\lfloor i/2 \rfloor$.
- The tree also should satisfy heap order property.
  - For every node X, the element stored at X should be smaller or equal to the element stored at the parent of X.





Insertion into a Heap

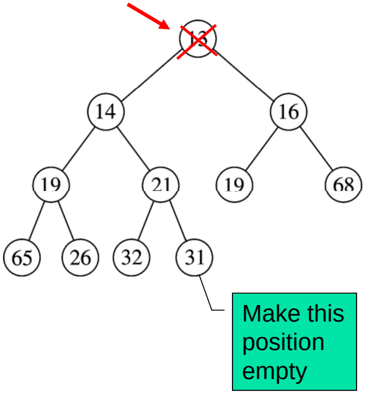


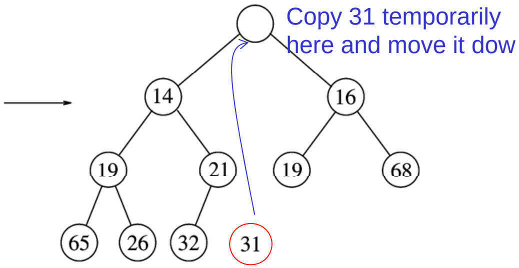
Insertion Algorithm

```
InsertToHeap(int x) {
        H[++n] = x;
        for(int k = n; k > 1; k /= 2) {
            if(H[k] > H[k / 2])
                swap(H[k], H[k / 2]);
            else
                break;
        }
}
```

- Since tree is a complete binary tree $h = O(\log n)$.
- Heapify procedure takes O($\log n$) time.

DeleteMIN from a Heap

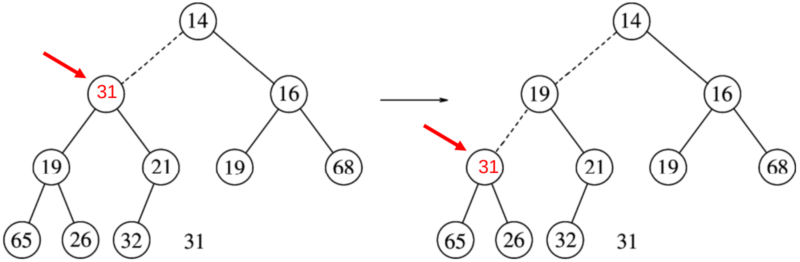




Is 31 > min(14,16)?

•Yes - swap 31 with min(14,16)

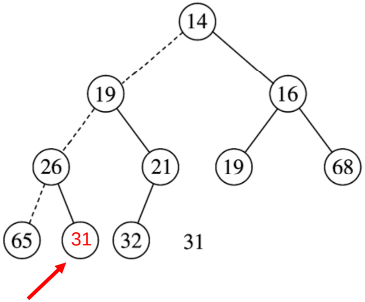
DeleteMIN from a Heap



Is 31 > min(19,21)?
•Yes - swap 31 with min(19,21)

Is 31 > min(65,26)?
•Yes - swap 31 with min(65,26)

DeleteMIN from a Heap



DeleteMIN Algorithm

```
DeleteMIN(HEAP H) {
    Last = H[n]; // Copy the last in temporary
    n = n-1;     // Update the last index
    i = 1;       // Starting index for heapify
    left = 2;    // Left child index
    right = 3;   // Right child index
    H[i] = Last; // Store last value into the root

    // Heapify in next slide
}
```

Heapify Algorithm

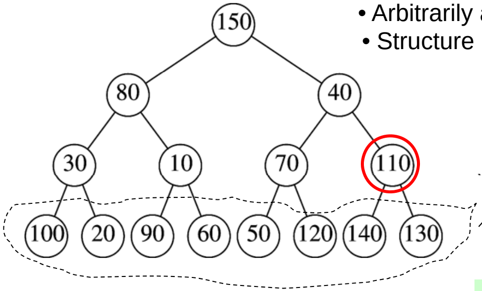
```
// Heapify operation
while (left ≤ n) {
        if (H[i] ≤ H[left] && H[i] ≤ H[right])
              return; // Heap property restored
        if (H[right] ≤ H[left]) {// swap with left
              swap(H[i], H[left]);
              i = left; // Heapify left subtree
        } else {// swap with right
              swap(H[i], H[right]);
              i = right; // Heapify right subtree
        }
        left = 2*i;
        right = left+1;
}
```

Building Heap

- A heap of n elements formed placing elements randomly into a binary tree.
- At leaf level (single element) heap property trivially holds.
- But heap property should be preserved at all levels.
- Pair up the heaps bottom up, by examining heap property for each internal node.
- Move the elements down from upper levels as needed.

Building Heap: Initial Configuration

Input: { 150, 80, 40, 10, 70, 110, 30, 120, 140, 60, 50, 130, 100, 20, 90 }



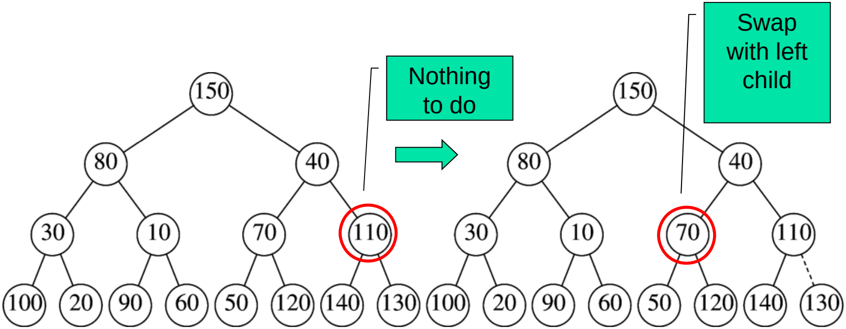
- Arbitrarily assign elements to heap nodes
- Structure property satisfied

Leaves are all valid heaps (implicitly)

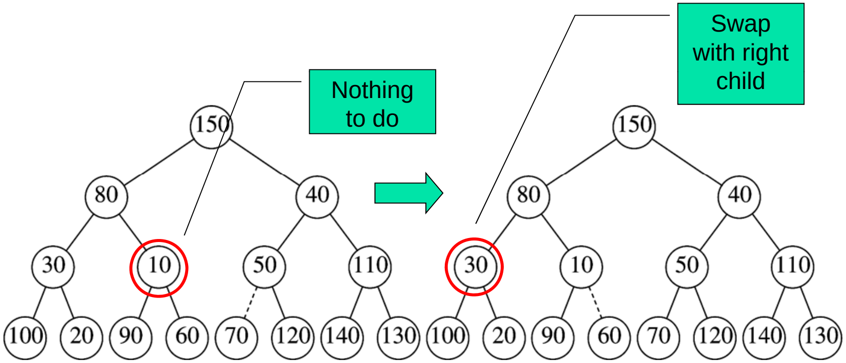
So, let us look at each internal node, from bottom to top, and fix it

- Heap order property violated
- Leaves are all valid heaps (implicit)

Building Heap: Intermediate Configuration

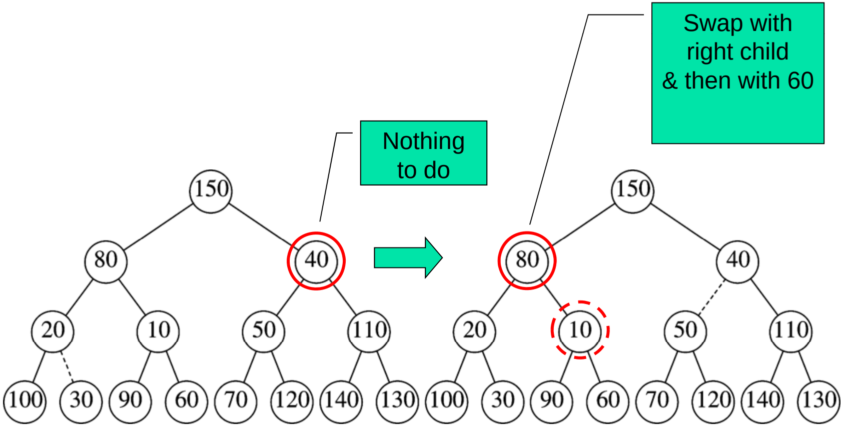


Building Heap: Intermediate Configuration



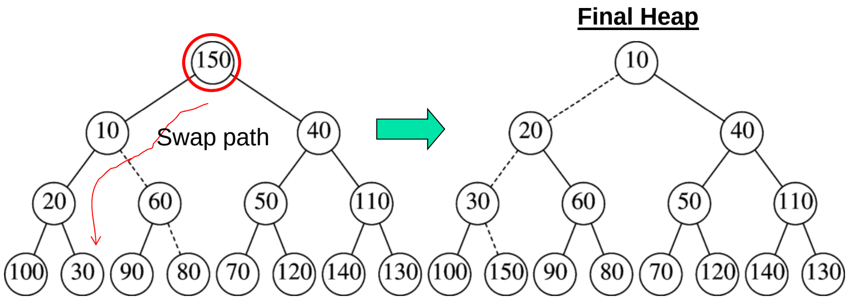
Dotted lines show path of percolating down

Building Heap: Intermediate Configuration



Dotted lines show path of percolating down

Building Heap: Final Configuration



Dotted lines show path of percolating down

Building Heap: Analysis

Time for Build Heap

Build heap takes time O(Sum of heights of all nodes).

Proof

- A node can go down from its own level along a tree path until it reaches a height h.
- At level ℓ there are 2^ℓ nodes, $0 \leq \ell \leq h$.

Therefore,

$$\begin{aligned}\sum_{\ell=o}^{h} 2^\ell \times (h-\ell) &= \sum_{\ell=o}^{h} 2^\ell \times h - \sum_{\ell=0}^{h} \ell \times 2^\ell \\ &= h(2^{h+1}-1) - S,\end{aligned}$$

where, $S = 1.2 + 2.2^2 + \ldots + h.2^h$.

Building Heap: Analysis

Evaluating S

$S = (h-1)2^{h+1} - 2$

Proof

$$\begin{aligned} 2S &= 1.2^2 + 2.2^3 + 3.2^4 + \ldots + (h-1)2^h) + h.2^{h+1} \\ 2S - S &= -(2 + 2^2 + 2^3 + \ldots + 2^h) + h.2^{h+1} \\ S &= -(2^{h+1} - 1 - 1) + h.2^{h+1} = (h-1)2^{h+1} - 2 \end{aligned}$$

Running time

Therefore, running time for building a heap is

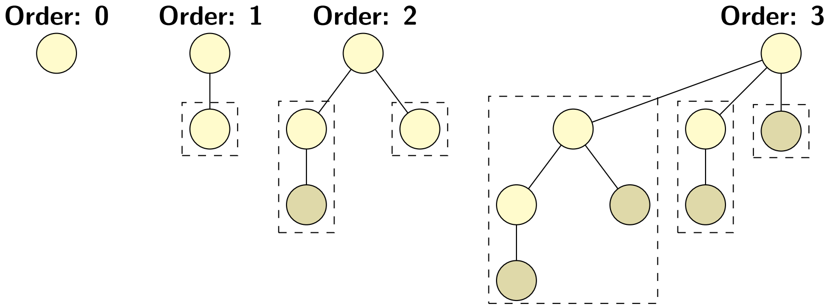
$$h.(2^{h+1} - 1) - (h-1)2^{h+1} + 2 = 2^{h+1} - h + 2.$$

Any Other Operations?

- Sometimes **increaseKey()** and **decreaseKey()** are also considered as basic operations on heaps.
- However, increase and decrease key operations require position of a key to be accessible in heap.
- Exposing the position of each key (i.e. internals of implementation) is equivalent to cheating in ADT approach,

Binomial Tree: Example

- Binomial tree is a kind of data structure than can be used for implementation of a heap ADT.
- Binomial tree is tree of height h is such that the number of nodes at level k is $\binom{h}{k}$, where $k \leq h$.



Binomial Heap

A Binomial heap is implemented used binomial trees.

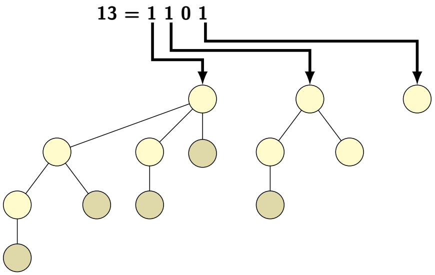
Properties of Binomial Heap

- Each key of a node is greater or equal to the key of its parrent.
- There can be just one or zero binomial trees of each order, including zeroth order.

- The second property implies that there can be at most $\log(n+1)$ binomial trees in the heap.

Example

- For example, 13 = 1101.
  - It has 1-tree of order 3 having 2^3 nodes.
  - It has 1-tree of order 2 having 2^2 nodes.
  - It has 1-tree of order 0 having 2^0 nodes.



A binomial heap having n nodes contains at most $\log(n+1)$ trees.

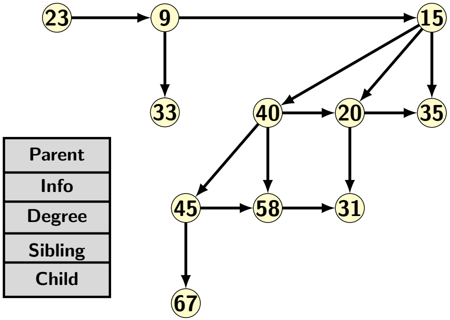
Operations on Binomial Heaps

1. Create a new Heap.
2. Search for minimum element.
3. Merge two binomial heaps.
4. Insertion of a new element.
5. Removal root of a tree.
6. Decreasing a key value.
7. Removal of an element .

Basic Properties of Binomial Heap

- A node is has 5 fields: `info, key, child, parent, sibling`
- Roots of the trees in a Heap are connected in a singly linked list via (`sibling` ptrs).
- The degree of a node represents number of children.
- The degrees of the roots in a heaps are strictly increasing from left to right.
- The minimum key is contained in the roots of $B_1, B_2, \ldots, B_k$.
- Let $N = b_{n-1}, \ldots b_2, b_1, b_0$, if $b_i = 1$ then heap contains binomial tree B_i.

Binomial Heap Representation



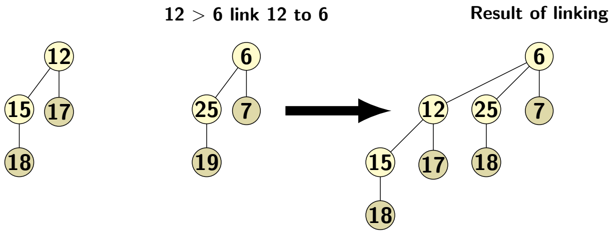
Linking Operation

- Most interesting operation is merge.
- The linking of two same order binomial trees is simple:
  - Assume there exists two binomial trees of order j
  - Compare keys in the roots of two order j trees.
  - Make the node with smaller key as the parent of the node with the larger key.

Linking Operation

```
linkBinomialTrees(T, S) {
    r1 = getRoot(T);
    r2 = getRoot(S);
    if (r1->key < r2->key) { // S is linked to T
        r2->parent = r1; r1->child = r2;
        r2->sibling = r1->child;
        r1->degree += 1;
     } else {  // T is linked to S
        r1->parent = r2; r2->child = r1;
        r1->sibling = r2->child;
        r2->degree += 1;
     }
     return T;
}
```

Example of Linking

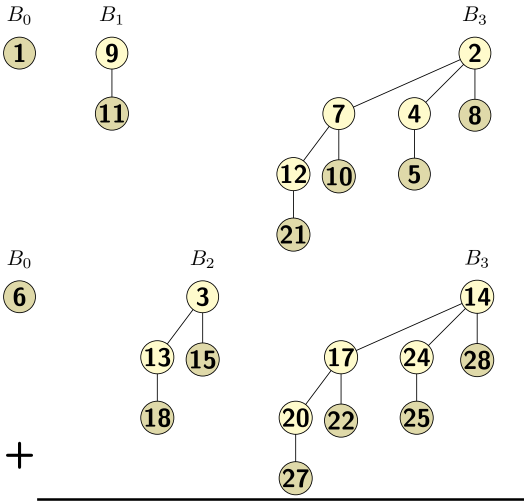


Merging Binomial Heaps

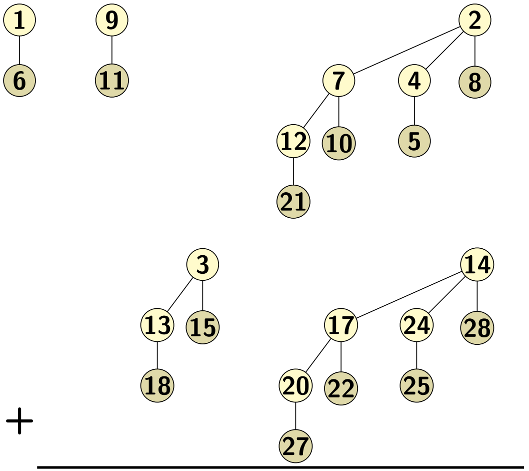
- Merging of heaps is complex, the number of trees varies and each tree correspond to a 1 bit position in n.
- Merging is carried out from right to left order just like adding a pair of binary numbers.
- When a carry occurs, it corresponds to merging of two binomial trees of same order.
- The addition example here has been shown in reverse order as the linked list of tree appear that way in our implementation.

carry:		B_1	B_2	B_3	
A_1:	B_0	B_1	–	B_3	
A_2:	B_0	–	B_2	B_3	
	–	–	–	B_3	B_4

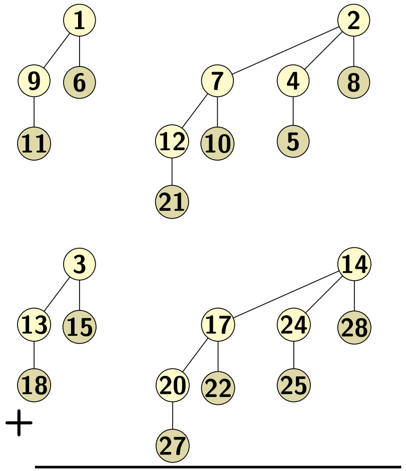
Example of Merging



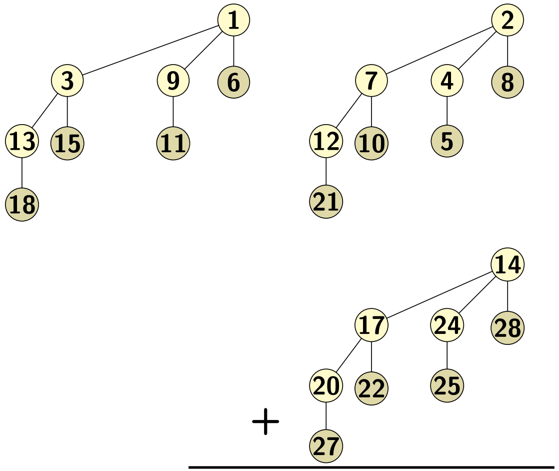
Example of Merging



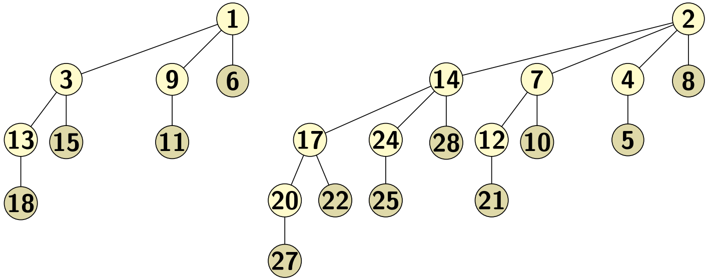
Example of Merging



Example of Merging



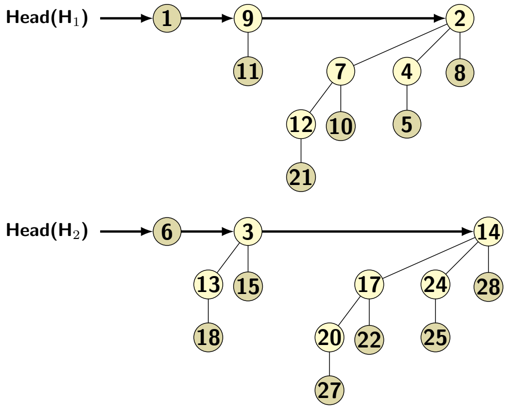
Example of Merging



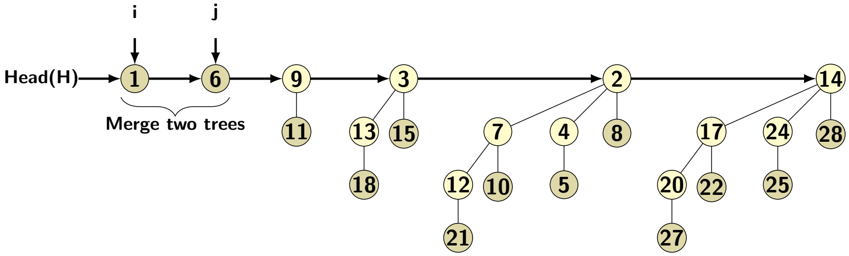
Union Operation

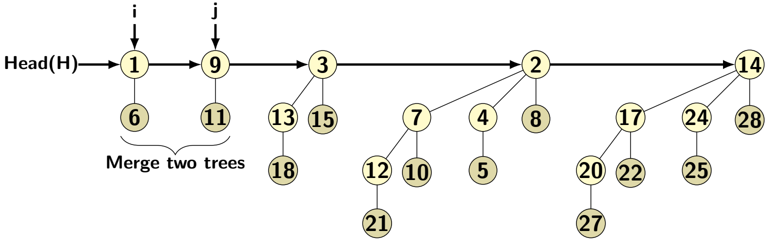
- It destroys both H_1 and H_2 and creates a new heap $H = H_1 \cup H_2$.
- It first uses a merging process to obtain H having all trees in monotonically sorted order in a linked list.
- Then linked list of roots is then traversed from left to right to merge the trees whenever possible.
- Let us focus on major steps of the process through the example once again.

Process of Union

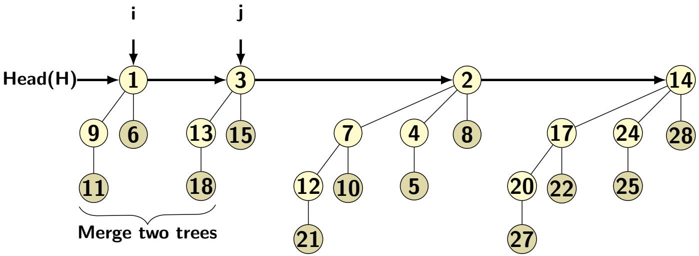


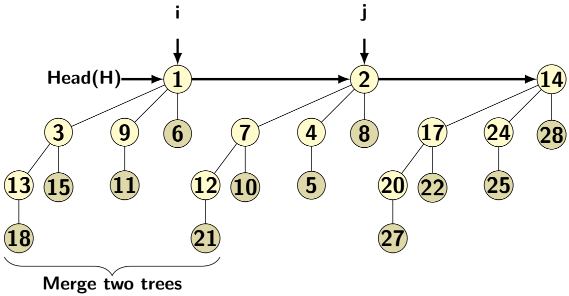
Creating List for $H = H_1 \cup H_2$





Merging Trees in H



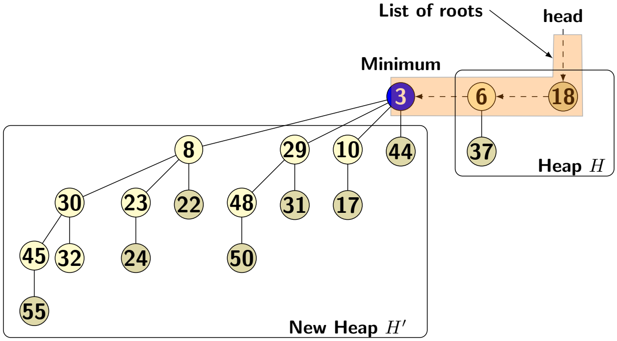


Binomial Heap Implementation

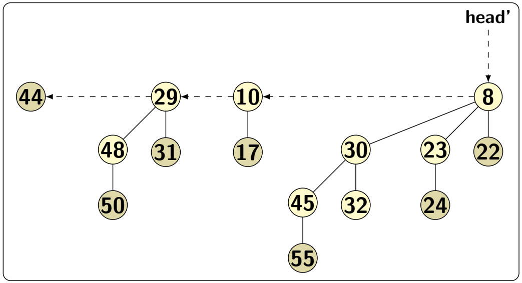
- Connect the roots of each tree in heap in a linked list.
- Let $head$ be the ptr to the first node.
- For **deleteMin()** operation, traverse list $head$ and find the minimum element `min`.
- Deletion of `min` creates a new heap H'
- Merge H and H'

DeleteMin

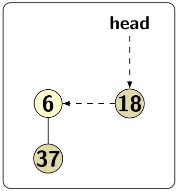
- Find root with minimum element in the root list.
- Deleting minimum breaks the concerned tree, call H'.



Merge H' and H

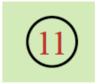


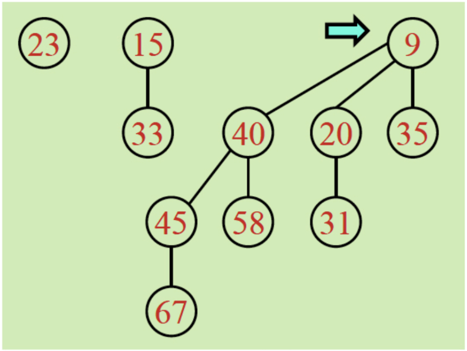
Heap H



Heap H'

Insert





New item is a one tree binomial heap

Binary Heap and Binomial Heap

Operation	Binary Heap	Binomial Heap
Create Heap	1	1
Insert	$\log n$	$\log n$
FindMIN	$\log n$	$\log n$
deleteMIN	$\log n$	$\log n$
Union	n	$\log n$
isEmpty	1	1

Heap Sort

- Building a heap of input value.
- Repeatedly perform deleteMIN and send the deleted item to the output.
- After every deleteMIN, heapify operation is applied to readjust the heap.
- So time for heap sort will consists of two parts:
  - O(n) for building the heap.
  - O($n \log n$) for n deleteMIN operation.
- Total time complexity is O($n \log n$).

Summary

- Learnt about binary heap and biomial heap.
- Heaps are useful for deleteMIN operations.
- In many cases we need to maintain priority queues for incoming tasks.
- Tasks have to be processed according to their priorities.
- Priority queues are implemented using heaps.
- Heaps are also useful in maintaining dynamic memory.
- Heaps are used servicing requests in order or priorities.
- Heap can be used for sorting.