Priority Queue: Heap Structures

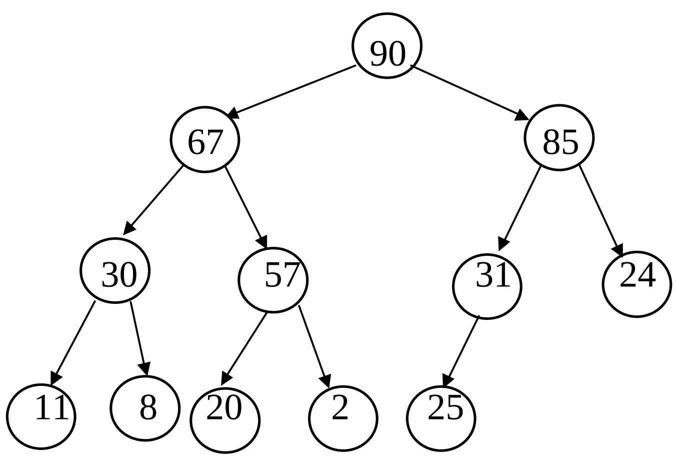
☞ **Definition:**

- A **max-heap** (**min-heap**) is a complete BT with the property that the value (priority) of each node is at least as **large** (**small**) as the values at its children (if they exist).

☞ **Implementation**:

- Sequential representation

☞ **Example**:



☞ **Operations:**

- Insertion
- Construct heap
- Deletion
- Delete_min (Delete_max)

☞ **Insertion of a heap**

- Procedure Insert(A[1..n],i)

/* Insert A[i] into the already hear A[1..n] */
Begin

While (I>1) and (A[i]> A[$\left\lfloor \frac{i}{2} \right\rfloor$]) do

Begin

swap(A[i], A[$\left\lfloor \frac{i}{2} \right\rfloor$]);

i = $\left\lfloor \frac{i}{2} \right\rfloor$;

Endwhile

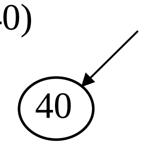
End;

- Example:

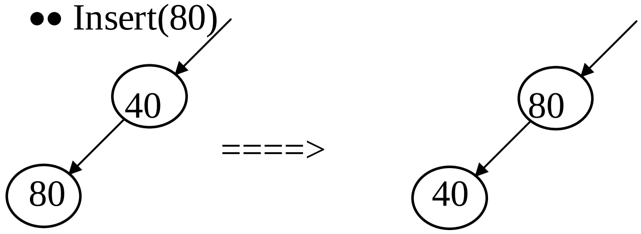
  - List of elements: 40, 80, 35, 90, 85, 100

  - The heap is empty

  - Insert(40)



  - Insert(80)



  - Etc.

☞ Construction of a heap

- **First method**:

•• Procedure construct_heap1(A[1..n])

```
/* The array will contain the heap */
Integer i;
Begin
        For I=2 to n do
                Insert (A[1..n],i);
        endfor;
end;
```

•• **Analysis**:

Theorem: Construct_heap1 takes O(nlogn) in the worst case.

Proof:

The worst case is when the elements are inserted in ascending order.

The Insert procedure takes O(longn).

Therefore, we have O(nlogn).

End proof.

•• **Note**: The best case when the elements are inserted in descending order. In this case The Insert procedure takes O(1). Therefore, Construct_heap takes O(n).

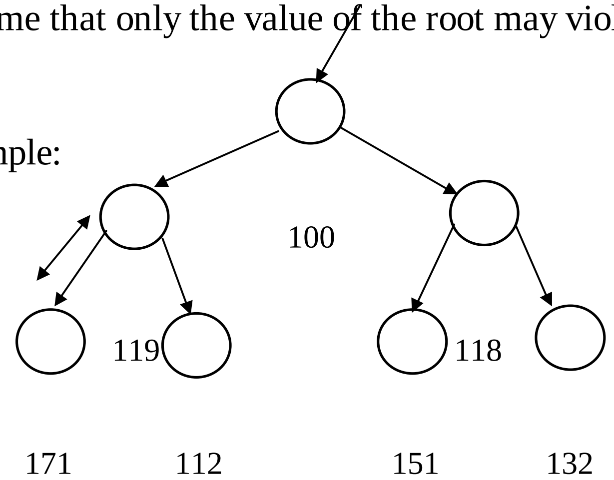
- **Second method**:

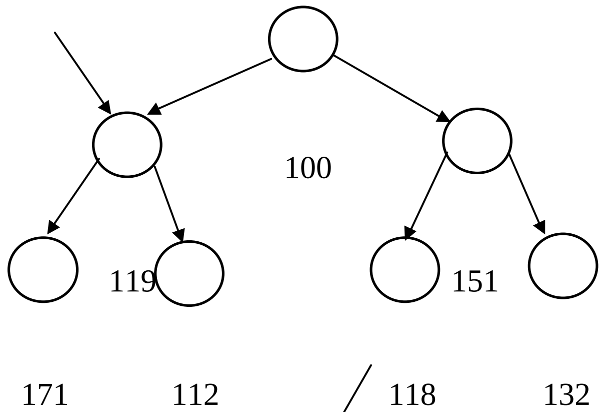
•• Takes the input array as a complete binary tree.

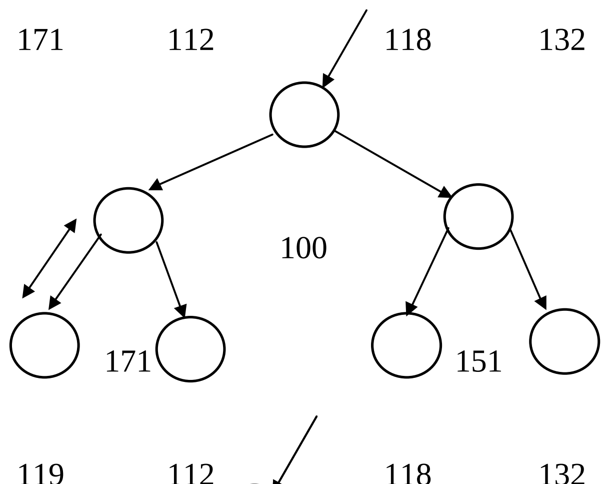
•• Construct the heap level by level from the leaves

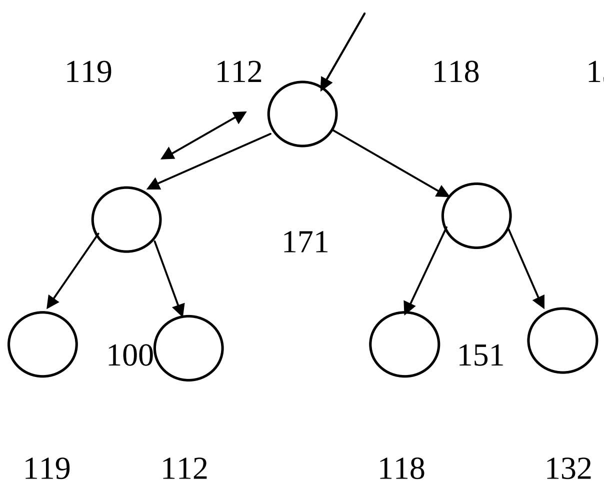
- Assume that only the value of the root may violate the heap property

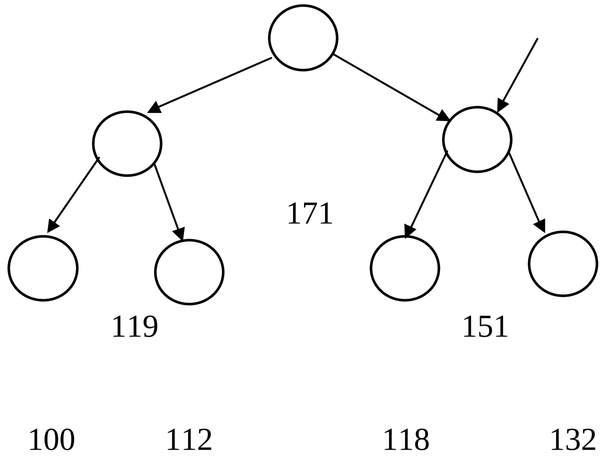
- Example:












```
•• Procedure Adjust_heap(A[1..n],i)
    /* A new heap is constructed from the value A[i] and the
/* heaps with roots A[2*i] and A[2*i+1] */
    Boolean done=false;
    type element;
    Begin
        j = 2*i; element = A[i];
        While ((j≤n) && (!done)) do
            /* Let j points to the largest child of A[⌊j/2⌋] */
            if ((j<n) and (A[j]< A[j+1]))
            then j = j + 1;
            endif;
            if (element ≥ A[j])
            then done = TRUE;
            else    begin
                        A[⌊j/2⌋] = A[j];
                        j = 2*j;
                    end;
            endif;
        endwhile;
        A[⌊j/2⌋] = element;
    End;
```

•• Procedure construct_heap2(A[1..n])
/* The array will contain the heap */
Integer i;
Begin

For i= $\left\lceil \frac{n}{2} \right\rceil$ to 1 step -1 do

Adjust_heap (A[1..n],i);

endfor;

end;

•• Analysis:

Lemma 1: There are at most $\left\lceil \frac{n}{2^{k-i+1}} \right\rceil$ nodes at level i in an n-element heap where $n = 2^k$.

Theorem: Construct_heap2 takes O(n) which is a tight bound.

Proof:

The total number of iteration of adjust-heap procedure is k-i for a node on level i, therefore, the total time, T, of Construct-heap2 is:

$$T = \sum_{1 \le i \le k} (k - i) \left\lceil \frac{n}{2^{k-i+1}} \right\rceil \qquad \text{Using Lemma 1}$$

Take j=k-i, we have 0≤j≤k-1.

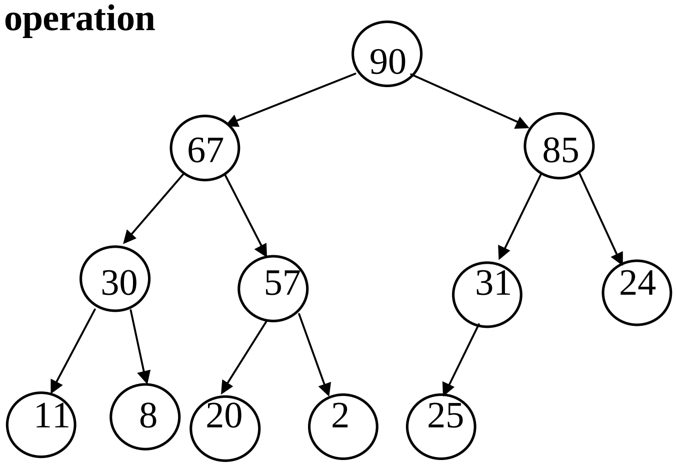
Hence,

$$T = \sum_{0 \le j \le k-1} j \frac{n}{2^{j+1}} = \frac{n}{2} \sum_{0 \le j \le k-} j 2^{-j}$$

Therefore, we have T = O(n).

End proof.

☞ **Delete operation**



- Take the content of the root out
- Put the last node in the heap in the root
- Adjust the heap.

☞ **Adjust the heap**:

- Procedure:

```
Procedure adjust_heap(A[1..n],i)
/* Move the last value in the heap to the root: i=1*/
Boolean done=false;
type element;
Begin
        j = 2i; element = A[i];
        While ((j≤n) && (!done))
        Begin
                /* j points to the largest child of A[⌊j/2⌋] */
                If ((j<n) && (A[j]<A[j+1]))
                then j = j + 1;
                endif;
                If (element ≥ A[j])
                then done = TRUE;
                else    begin
                                A[⌊j/2⌋] = A[j];  j = 2*j;
                        end;
                endif;
        Endwhile;
        A[⌊j/2⌋] = element;
end;
```

- Complexity:

  •• **O(logn)** where n is the number of elements in the heap.

Sorting: HeapSort

☞ Motivation:

- The worst case is O(nlogn)

☞ procedure:

- Procedure

```
Procedure Heapsort(A[1..n])
int i;
Begin
    construct_heap2(A[1..n])
    for i=n to 2 step -1 do
        swap(A[1], A[i]);
        Adjust_heap((A[1..(i-1)]))
    endfor;
end;
```

- Complexity:

  - •• Let n be the number of element to be sorted.
  - •• Heap construction takes O(n)
  - •• Adjust heap takes O(logn)
  - •• The for loop takes O(n)

    •• Therefore, O(nlogn).

Sets and Disjoint Set Union

☞ **Set Representations**

Bit map or Characteristic vector

I º Ù

Y º Ú

Disadvantages: small set and large value of objects
(Universal set)

Trees

☞ **Disjoint sets**

Definition:

A disjoint set data structure maintains a collection of S of disjoint dynamic sets

Operations:

Union: S_i & S_j

Find(i): Find the set containing the element i.

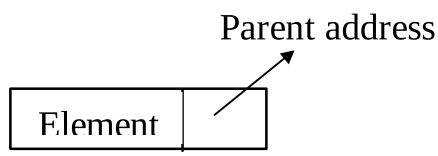
Problem:

Develop an efficiency data structure and algorithms to perform a linear sequence of Unions and Finds

Representation:

Sets = Trees
Name of a set is the root of the tree.

Each node:



The root node has a parent field of 0

☞ First Algorithm

Union:

```
Procedure Union(i,j);
Begin
        parent (i) = j;
end;
```

Complexity: O(1)

Find:

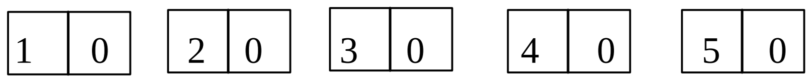
```
Function Find(i);
Integer j;
Begin
        j=i;
        While parent(j) <> 0 do
                j = parent(j);
        endwhile;
        return(j);
end;
```

Complexity: O(n);

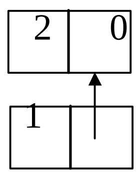
Analysis: Worst Case Behavior of Union & Find Algorithms:

Given U(1,2), F(1), U(2,3), F(1), U(3,4), F(1), U(4,5), F(1)

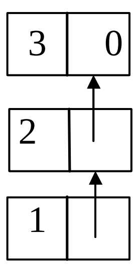
- Initialization:



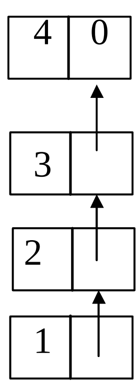
Union (1,2): Find(1) takes 1



Union (2,3): Find(1) takes 2



Union (3,4): Find(1) takes 3



Etc.

For a sequence of n Unions and n Finds, the total number of operations is:

$$n + 1 + 2 + 3 + ... + (n-1) = \frac{\mathbf{n(n+1)}}{\mathbf{2}} \implies O(n^2).$$

Amortized running time is: O(n).

☞ Weighting Rule Algorithm

Same Find algorithm

Modified UNION:

Union(i,j): If the number of nodes in tree i is less than the number of nodes in tree j, then make j the parent of i, otherwise make I the parent of j.

Implementation: Use the parent field of the root as a counter.
Parent field contains the number of elements in th etree (negative).

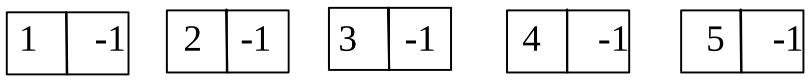
```
Procedure Union(i,j);
/* Tree with less nodes becomes the parent */
integer x;
Begin
        x = parent(i)+parent(j);
        If (parent(i)>parent(j))
        then
                parent (i) = j;
                parent(j) = x;
        else
                parent(j) = i;
                parent(i) = x;
        endif;
end;
```

Analysis: O(1);

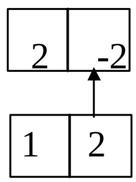
Analysis: Worst Case Behavior of Union & Find Algorithms:

Given U(1,2), F(1), U(2,3), F(1), U(3,4), F(1), U(4,5), F(1)

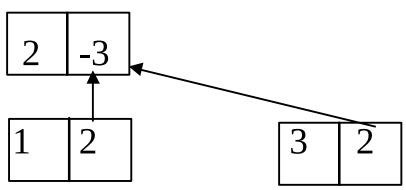
- Initialization:



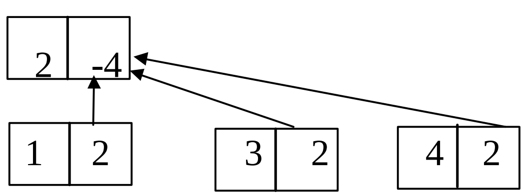
Union (1,2): Find(1) takes 1



Union (2,3): Find(1) takes 1



Union (3,4): Find(1) takes 1



Etc.

For a sequence of n Unions and n Finds, the total number of operations is:

$$n + 1 + 1 + ... + 1 = 2n-1 \quad ===> \; O(n).$$

Amortized running time is: O(1).