Chapter 7: Priority Queues

What are we studying in this chapter?

- Single and double ended priority queues
- Leftist trees
- Binomial heaps
- Fibonacci heaps
- Pairing heaps **- 6 hours**

7.1 Introduction

In this section, let us see another variation of queue called priority queue. Let us see "What is priority queue?"

Definition: A priority queue is a collection of elements such that each element has an associated priority. The elements can be inserted arbitrarily but the elements have to be deleted based on the priority. Always an element with highest priority is processed before processing any of the lower priority elements. If the elements in the queue are of same priority, then the elements are processed based on first come first service. **Note:** Heaps are used to implement priority queues.

For example, priority queues are used in job scheduling algorithms in operating system where the jobs with highest priorities have to be processed first.

Now, let us see "What are the various types of priority queues?" The priority queues are mainly classified as shown below:

- Single ended priority queue
- Double ended priority queue
- Meldable priority queues

7.1.1 Single ended priority queues

Now, let us see "What is a single-ended priority queue?"

Definition: A single-ended priority queue is a data structure that supports following operations:

- Return an element with minimum (maximum) priority
- Insert an element with arbitrary priority
- Delete an element with minimum (maximum) priority

7.2 💻 Priority queues

Single ended priorities are classified into two categories:

- Ascending priority queue (min priority queue): The various operations that are supported by min priority queue are:
  1. Insert an element with arbitrary priority: If n represent number of elements, this operation can be performed in $\log n$ unit time denoted by O($\log n$)
  2. Return an element with minimum priority: Minimum element can be found in 1 unit of time denoted by O(1)
  3. Delete an element with minimum priority: If n represent number of elements, this operation can be performed in $\log n$ unit time denoted by O($\log n$)

- Descending priority queue (max priority queue): The various operations that are supported by max priority queue are:
  1. Insert an element with arbitrary priority: If n represent number of elements, this operation can be performed in $\log n$ unit time denoted by O($\log n$)
  2. Return an element with maximum priority: Maximum element can be found in 1 unit of time denoted by O(1)
  3. Delete an element with maximum priority: If n represent number of elements, this operation can be performed in $\log n$ unit time denoted by O($\log n$)

7.1.2 Double ended priority queues

Now, let us see "What is a double-ended priority queue?"

Definition: A double-ended priority is a data structure that supports operations of *min priority queue* and *max priority queue*. The various operations that are performed by double-ended priority queue are:

- Return an element with minimum(least) priority
- Return an element with maximum(highest) priority
- Insert an element with arbitrary priority
- Delete an element with minimum(least) priority
- Delete an element with maximum(highest) priority

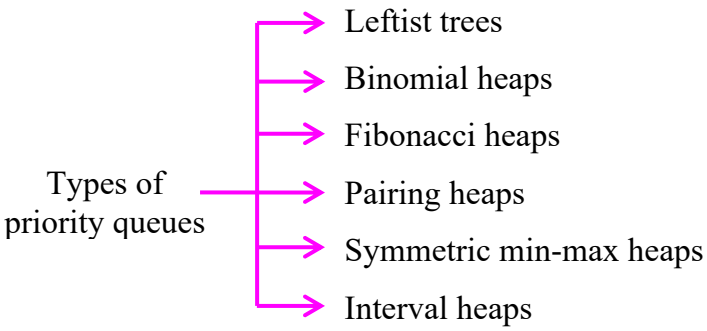
7.1.3 Meldable priority queues

The single ended priority queue can be extended to implement meldable (single-ended) priority queue. Now, let us see "What is a meldable priority queue?"

Definition: The meldable priority queue is an extension of single-ended priority queue which melds two priority queues together. The two data structures that are used for implementing meldable priority queues are leftist trees and binomial heaps.

A meldable priority queue also includes operations to delete an arbitrary element and to decrease the priority of an element. The two data structures that are used for this purpose are Fibonacci heaps and pairing heaps.

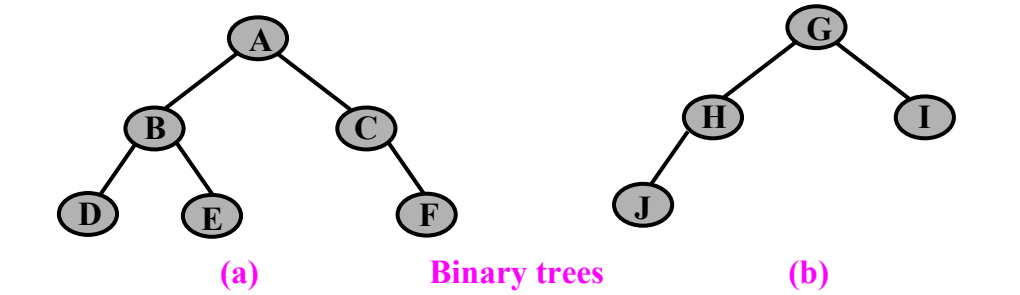
Now, let us see "What are the different types of meldable priority queues?" The meldable priority queues are classified as shown below:



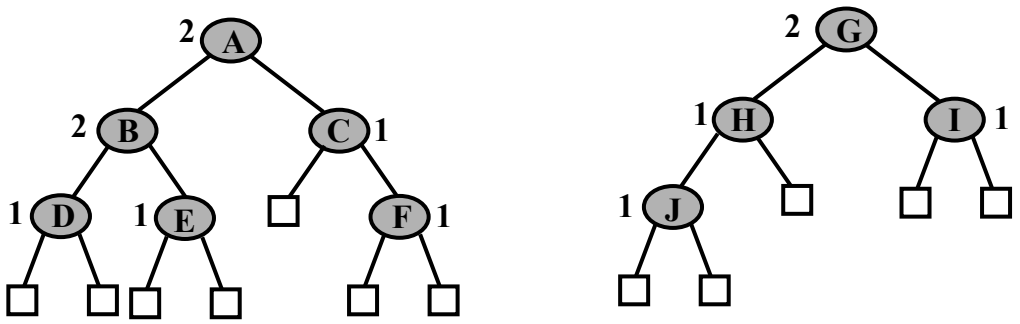
7.2 Leftist trees

The leftist trees are defined using the concept of an extended binary tree. Now, let us see "What is an extended binary tree?"

Definition: A binary tree in which all empty children are replaced by square nodes is called an extended binary search tree. For example, consider the binary trees without external nodes (fig. a and b.) and its equivalent tree with external nodes (fig. c and d):



(a) **Binary trees** (b)



(c) **Extended binary trees** (d)

Note: The square nodes in an extended binary tree are called external nodes, rest are called internal nodes.

- Let *x* be an arbitrary node in an extended binary tree
- LeftChild(x) denote the left child of internal node *x*
- RightChild(x) denote the right child of internal node *x*

Now, let us "Define the term shortest(x)"

Definition: shortest(x) is recursively defined as shown below:

$$\text{shortest}(x) = \begin{cases} 0 & \text{if x is external node} \\ 1 + \min\{\text{shortest}(\text{LeftChild}(x)), \text{shortest}(\text{RightChild}(x))\} & \text{otherwise} \end{cases}$$

For example, the numbers written by the side of each node in the trees shown in fig(c) and fig(d) in previous page is the value of *shortest(x)* and can be computed as shown below:

shortest(D) = 1 + min { shortest(leftchild(D), shortest(rightchild(D)) }
= 1 + min { 0 , 0 }
= 1

shortest(C) = 1 + min { shortest(leftchild(C), shortest(rightchild(C)) }
= 1 + min { 0 , 1 }
= 1

shortest(B) = 1 + min { shortest(leftchild(B), shortest(rightchild(B)) }
= 1 + min { 1 , 1 }
= 2

shortest(A) = 1 + min { shortest(leftchild(A), shortest(rightchild(A)) }
= 1 + min { 2 , 1 }
= 2

Now, let us see "What are the different types of leftist trees?" The two types of leftist trees are:

- Height-Biased Leftist Trees (HBLT)
- Weight-Based Leftist Trees (WBLT)

7.2.1 Height-Biased Leftist Tree

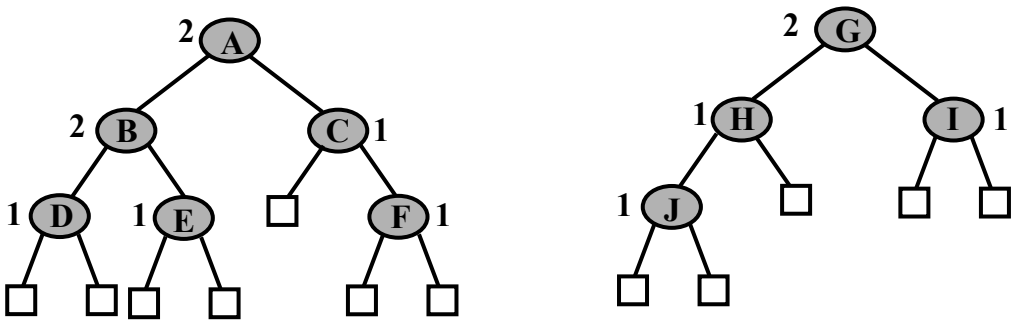
Now, let us see "What is height biased leftist tree (or leftist trees)"

Definition: A leftist tree (also called height biased leftist tree) is a non-empty binary tree that satisfies the following condition:

shortest(LeftChild(x)) ≥ shortest(RightChild(x))

for any given node *x*. Traditionally, height biased leftist tree is also called leftist tree. Since the information of shortest(LeftChild(x)) is greater than or equal to shortest(RightChild(x), the name leftist is selected.

For example, consider the following external binary trees along with shortest(x) for any node x:



(a) Non-leftist tree **(b) Leftist tree**

Ex 1: The tree shown in figure (a) is not a leftist tree. This is because, for any tree to be leftist, the following condition should be satisfied:

shortest(LeftChild(x)) ≥ shortest(RightChild(x))

Let x = C. Now, 0 ≥ 1 is **false.** So, it is not a leftist tree

Ex 2: The tree shown in figure (b) is a leftist tree. This is because, for any tree to be leftist, the following condition is satisfied:

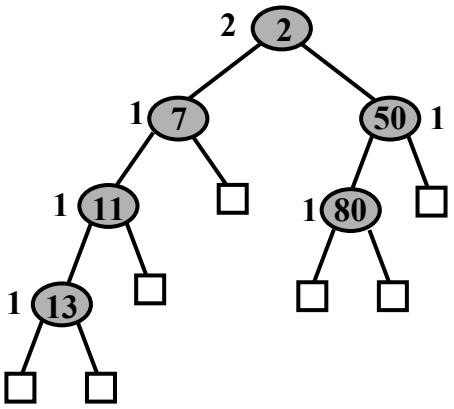
shortest(LeftChild(x)) ≥ shortest(RightChild(x))

Now, let us see "What are the different types of leftist trees?" The two types of leftist trees are:

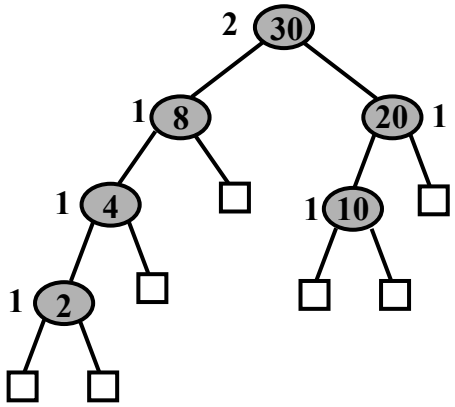
7.6 💻 Priority queues

- **Min leftist tree:** A *min leftist tree* is a leftist tree in which each key value in each node is less than their children.
- **Max leftist tree:** A *max leftist tree* is a leftist tree in which each key value in each node is greater than their children.

For example, the tree shown in figure (a) is *min leftist tree* and the tree shown in figure (b) is *max leftist tree*.



(a) min leftist tree

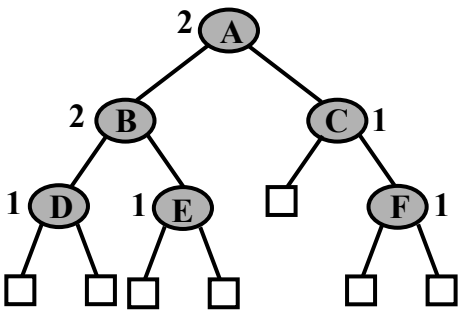


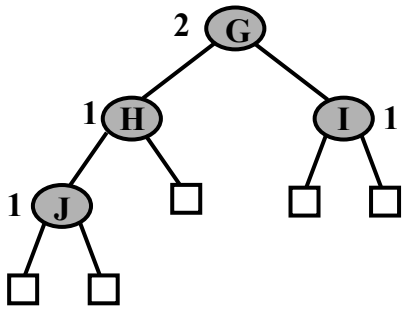
(b) max leftist tree

7.2.1 Weight-Biased Leftist Tree

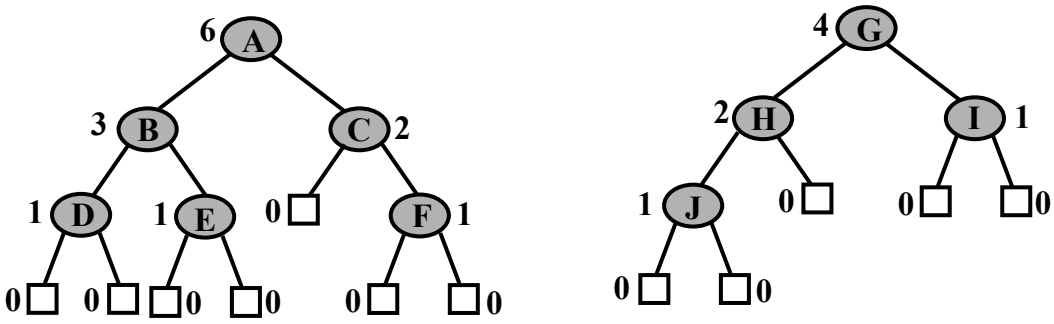
There is another kind of leftist tree by considering the number of nodes in the subtree, rather than the length of shortest root to external node path called weight-biased leftist tree. Now let us "Define weight w(x) of a node"

Definition: The weight w(x) of a node *x* is the number of internal nodes in the subtree with respect to root *x*. Note that if *x* is external node, its weight is 0. If *x* is internal node, then its weight w(x) = sum of weights of its children + 1. For example, consider the following external binary trees:





For example, the numbers written by the side of each node in the trees shown below is the value of *w(x)*.



The weights of each node in the above tree can be calculated as shown below:

Note: The weight w(x) = 0 for all external nodes i.e., all square nodes have weight 0. The weights of internal nodes can be calculated as shown below:

- w(D) = w(LeftChild(D)) + w(RightChild(D)) + 1
  = 0 + 0 + 1
  = 1

  On similar lines w(E) = w(F) = w(J) = w(I) = 1

- w(B) = w(LeftChild(B)) + w(RightChild(B)) + 1
  = 1 + 1 + 1
  = 3

- w(C) = w(LeftChild(C)) + w(RightChild(C)) + 1
  = 0 + 1 + 1
  = 2

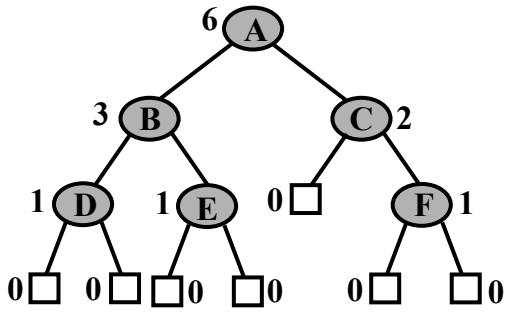
- w(A) = w(LeftChild(A)) + w(RightChild(A)) + 1
  = 3 + 2 + 1
  = 6

  On similar lines w(H) = 2 and w(G) = 4

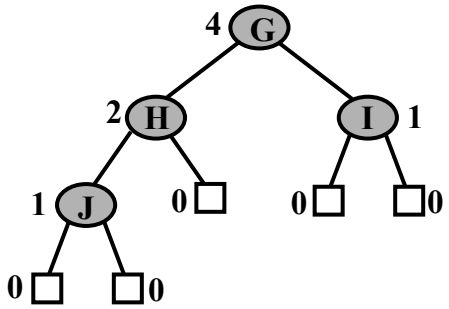
Now, let us see "What is weight biased leftist tree?"

7.8 💻 Priority queues

Definition: A binary tree in which the weight *w* of every left child is greater than or equal to weight of right child is called weight-biased leftist tree **(WBLT).** For example, consider the two extended binary trees with weights shown below:



(a) Not a weight-biased leftist tree



(b) weight-biased leftist tree

Ex 1: For a WBLT, the condition to be satisfied is:

w(LeftChild(x)) >= w(RightChild(x))

If x = C, then w(LeftChild(C)) >= w(RightChild(C))

But, 0 >= 1 is false.

So, the tree shown in fig(a) is not a weight-biased leftist tree.

Ex 2: The tree shown in figure (b) is a weight-biased leftist tree. This is because, for each node in binary tee, the following condition is satisfied:

$$w(\text{LeftChild}(x)) \geq w(\text{RightChild}(x))$$