A (Very) Brief Introduction to Binary Heaps and Binary Trees

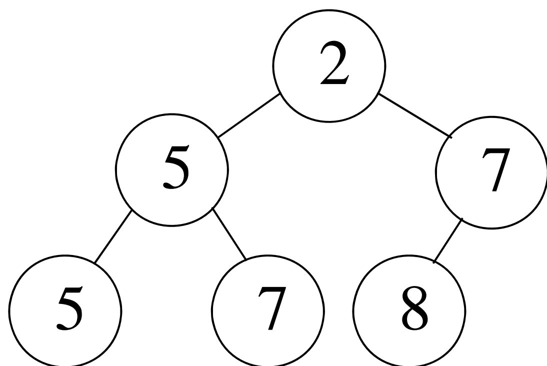
A Primer for Lab 4

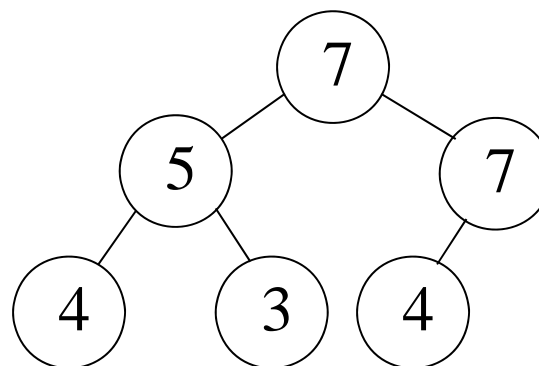
Full Binary Tree: Each node has exactly 0 or 2 children. Each internal node has exactly 2 children; each leaf has 0 children.

Complete Binary Tree: A full binary tree where all leaves have the same depth.

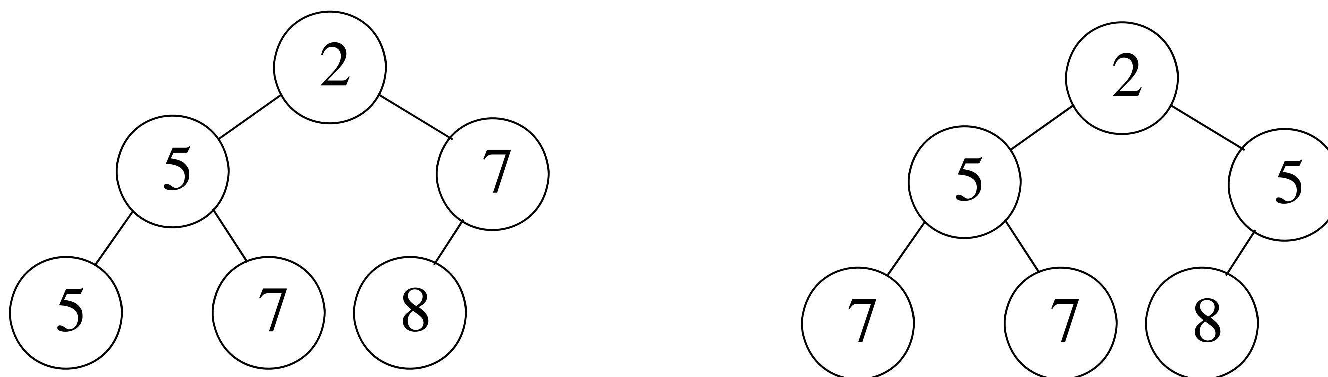
Nearly Complete Binary Tree: All leaves on the last level are together on the far left side, and all other levels are completely filled. (See the next slide.)

A **minimum binary heap** (LHS below) is a *nearly complete binary tree* such that the data in every node is less than or equal to the data in each of its child nodes and the node's left and right sub-trees are also minimum heaps. (There's also a **maximum binary heap** (RHS below).)



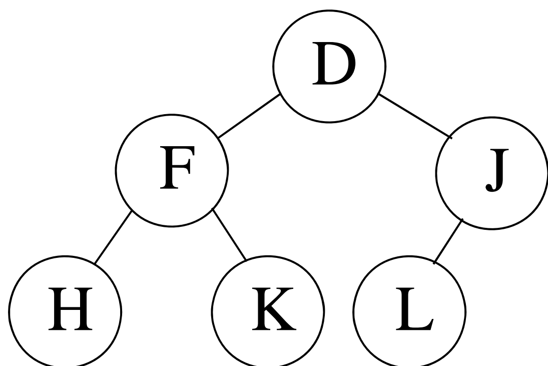


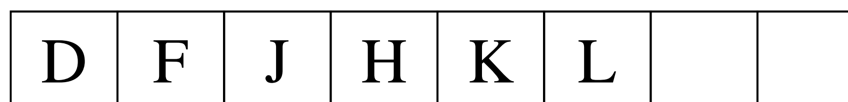
It is important to realize that two binary heaps can contain the same data but the data may appear in different positions in the heap:



Both of the minimum binary heaps above contain the same data: 2, 5, 5, 7, 7, 8. Even though both heaps satisfy all the properties necessary of a minimum binary heap, the data is stored in different positions in the tree.

The fact that a binary heap is a *nearly complete* binary tree allows us to represent the heap using an array. Hence we have a *contiguous* rather than *linked structure* to represent the heap.





We must now determine a way of navigating the heap. If the heap were represented as a linked structure, each node would have a pointer to its left and right child. Using the array representation, a node with index k in the array has:

index of left child: $2k + 1$

index of right child: $2k + 2$

index of parent: floor((k-1)/2)

Example of a Binary Tree Node in C++

We will assume that a `typedef` statement has defined `Item_type` to be the type of data to be stored in the tree. Each node contains an item, a pointer to the left subtree and a pointer to the right subtree:

```
typedef string Item_type;

struct BNode
{
    Item_type item; // may have many data fields
    BNode*    left;
    BNode*    right;
};
```

You can also use a class.

To begin, we will write a `makeNode` function to create a new `BNode` as needed. Note the default assignments in the argument list …

```
Bnode * makeNode(const Item_type& item,
    Bnode * leftChild = NULL, Bnode * rightChild = NULL)
// PRE:  item is valid, leftChild points to a node or is
//       NULL, rightChild points to a node or is NULL
// POST: a new node is created and its address is
//       returned
{
   Bnode * temp;
   temp = new Bnode;
   temp->item = item;
   temp->left = leftChild;
   temp->right = rightChild;
   return temp;
}
```