CMSC 420: Lecture 5
Priority Queues and Heaps

Priority Queues: A *priority queue* is an abstract data structure storing key-value pairs. The basic operations involve inserting a new key-value pair (where the key represents the *priority*) and extracting the entry with the smallest priority value. These operations are called *insert* and *extract-min*, respectively. In contrast to a standard queue (first-in, first-out) a priority queue extracts entries according to their priority. As an example, irrespective of the order in which passengers arrive at the gate, airlines often board them according to row number from the rear of the plane.

Priority queues can be implemented in many different ways. For example, you could maintain a simple linear list of key-priority pairs. But how are these to be sorted? If you sort by arrival order, then insertion is fast, but extraction requires checking all the priorities, which takes $O(n)$ time. On the other hand, if you sort by priority, extraction is fast, but insertion involves determining where to put the new item, and this (naively) takes $O(n)$ time.

The question is whether it is possible to achieve both operations in $O(\log n)$ time. There is a collection of related tree-based data structures that support these times. Because they share the same general structure, they are all called *heaps*.

Heaps: At its most generic, a *heap* is a rooted (typically binary) tree where each node stores a key-value pair. These data structures all have one common aspect, other than the root node, the priority of every node is greater than or equal to its parent (see Fig. 1). A tree that satisfies this property is said to be in *heap order*. (This is sometimes called a *min heap*. Reversing the order results in a *max heap*. The max heap is usually used in HeapSort.)

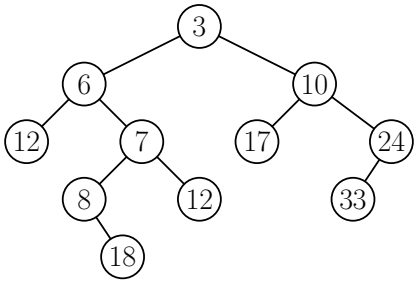

Fig. 1: A (min) heap-ordered binary tree. (Only priorities are shown.)

Note that in such a tree, the smallest priority item is always at the root. They main questions are how to maintain this structure as elements are inserted and extracted.

There are many variants of the heap data structure. If you learned about the sorting algorithm called *HeapSort*, then you no doubt learned the the simplest of these structures, called the *binary heap* (which we will present below). But when additional operations are desired (for example, altering individual priority values or merging two heaps together), there are alternative data structures that are more efficient than the binary heap. This has given rise to data structures with various esoteric names such as *binomial heaps*, *Fibonacci heaps*, *pairing heaps*, *quake heaps*, *leftist heaps*, and *skew heaps*. (For further information, see this Wikipedia article on Heaps.) In this lecture, we will discuss just a couple of these, standard binary heaps and leftist heaps.

Binary Heaps: The data structure used in HeapSort is called a *binary heap*. It is a venerable data structure, invented by J. W. J. Williams way back in 1964. It has a number of very elegant features. Most notably, even though it is a binary tree, it can be stored in an array, without the needs for dynamic memory management or pointers.

Recall from our lecture on trees that a binary tree is *complete* (sometimes called *left-complete*) if every level of the tree is completely filled, except possibly the bottom level, which is filled from left to right. It is easy to verify that a complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes, implying that a tree with n nodes has height $O(\log n)$ (see Fig. 2). (We leave these as exercises involving geometric series.)

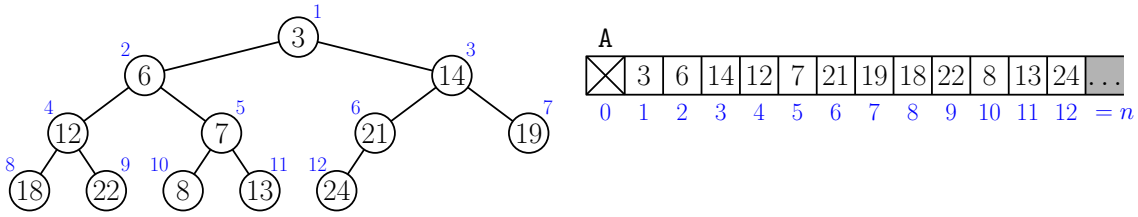

Fig. 2: A heap-ordered complete binary tree and its mapping to an array.

The regular structure allows us to use arithmetic to identify tree relations. Given a complete tree with n elements stored in an array `A[1..n]`, and for any index $1 \leq i \leq n$, we can access its immediate relations:

- left(i): if ($2i \leq n$) then $2i$, else `null`
- right(i): if ($2i + 1 \leq n$) then $2i + 1$, else `null`
- parent(i): if ($i \geq 2$) then $\lfloor i/2 \rfloor$, else `null`

Note that we are effectively wasting element `A[0]` by using this scheme. It is possible to modify the indexing rules to start with index zero, but this makes the access formulas a bit more complicated.

Binary Heap Insertion: Let us assume that our binary heap currently contains n elements and is stored in the array `A[1..n]`. (To initialize the structure, we simply set $n \leftarrow 0$.) To insert a new key `x` into the binary heap, we increment n and add the new item at the end of the array. We then "sift" the new key up the tree by swapping it with its parent as long as its priority is smaller than its parent, or until hitting the root, whichever comes first (see Fig. 3).

The insertion code is presented in the following code block. The code is slightly different from our description. To make the code a bit faster, rather than storing the new key and swapping, we copy entries down and insert the new key at the end. Clearly, the running time is proportional to the height of the tree, which we have shown is $O(\log n)$.

Binary Heap Extract-Min: To perform extract-min, we already observed that the minimum element is at the root. But, if we remove this element, we have a hole that needs to be filled. There is an elegant method to fill this hole. We first save the root element as our final result. We then copy the nth element of the array to index 1 and decrement the value of n. Finally, we sift the new root element down the tree to restore the heap property. To sift an element down, we first determine which of its two children has the smaller priority. We compare the current node with this smaller child. If the child has a smaller priority than the current node,

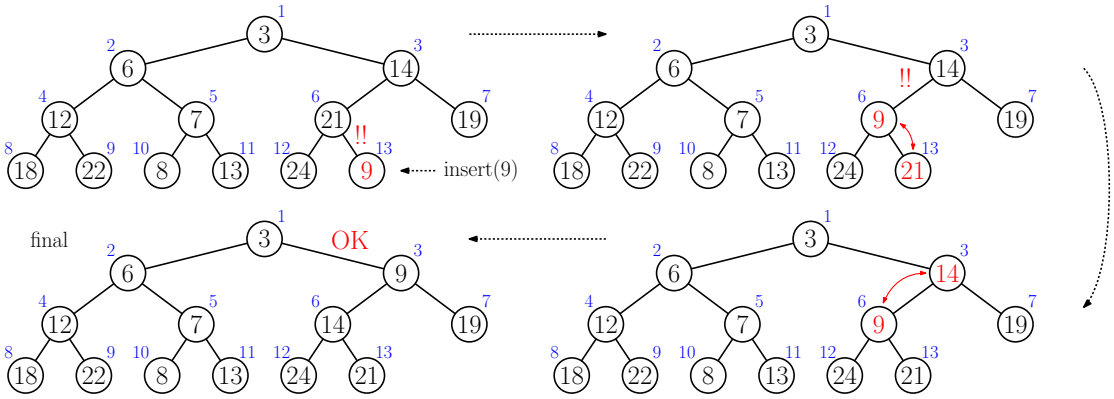

Fig. 3: Example of inserting a new element (9) into a binary heap and sifting up.

Binary-Heap Insertion

```
void insert(Key x) {                    // insert new key x in heap
    n += 1                              // increment tree size
    i = sift-up(n, x)                   // sift x up the tree to its final position
    A[i] = x                            // store x in the heap
}

int sift-up(int i, Key x) {             // sift x up to its proper position
    while(i > 1 && x < A[parent(i)]) {  // x's parent exists and is too large?
        A[i] = A[parent(i)]             // move parent down (and x up)
        i = parent(i)
    }
    return i
}
```

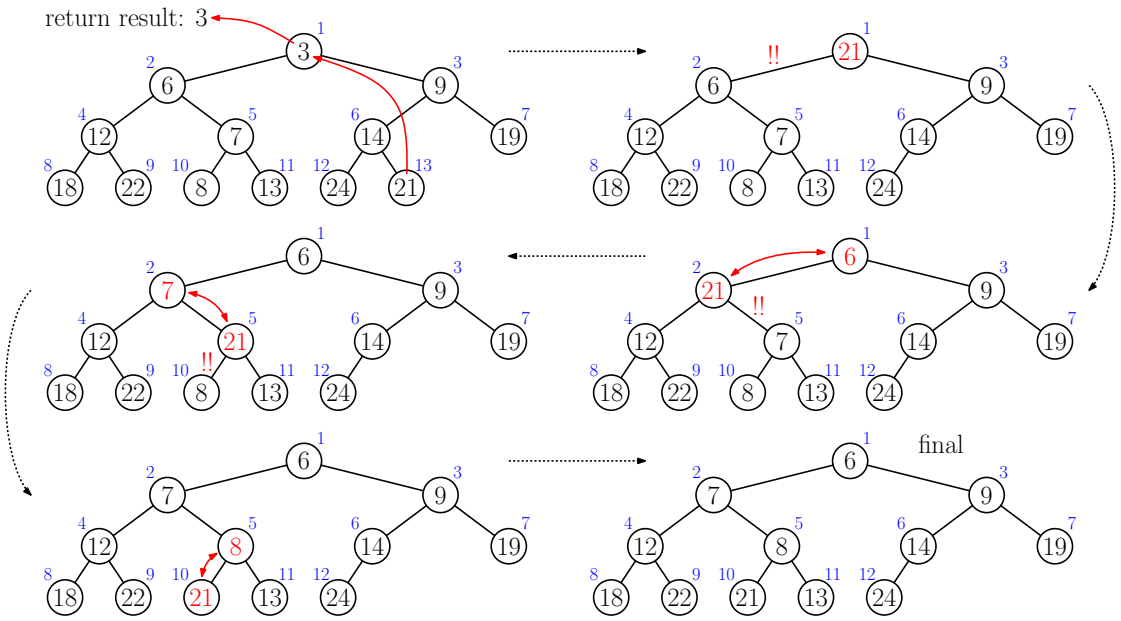

Fig. 4: Example of extract-min. We save the root element (3) as the final result, and copy the last element (21) into the root and decrement n. We then repeatedly sift it down by swapping it with the smaller of its two children until either reaching the leaf level or until both of its children are at least as large.

Binary-Heap Extract-Min

```
Key extract-min() {                         // extract the minimum from heap
    if (n == 0) Error - Empty heap
    Key result = A[1]                       // save final result
    Key z = A[n--]                          // get element to sift (and decrement n)
    i = sift-down(1, z)                     // sift z from root to its final position
    A[i] = z                                // move z into its proper position
    return result
}

int sift-down(int i, Key z) {               // sift z down to its proper position
    while (left(i) != null) {               // repeat until arriving at a leaf
        u = left(i);  v = right(i)          // i's left and right children
        if (v != null && A[v] < A[u])       // swap so that u has smaller key
            u = v
        if (A[u] < z) {                     // sift z by moving A[u] up
            A[i] = A[u]
            i = u
        } else break                        // done sifting
    }
    return i                                // return z's proper position
}
```

then we swap them. We repeat this along the we have a child and the swap takes place. The process is illustrated in Fig. 4) and the code is presented just below.

Meldable Heaps: The standard binary heap data structure is an simple and efficient data structure for the basic priority queue operations insert and extract-min. Suppose that we have an application in which in addition to insert and extract-min, we want to be able to *meld* or *merge* the contents of two different queues into one queue. As an application, suppose that we have a two-processor computer system and each processor has a priority queue of processes waiting to be executed. If one of the processors fails, we need to merge the two queues so that the remaining processor can handle all of them.

Let's introduce a new operation `H = merge(H1, H2)`, which takes two existing priority queues H_1 and H_2, and merges them into a new priority queue, H. This operation is *destructive*, which means that the priority queues H_1 and H_2 are destroyed in order to form H.

We would like to be able to implement `merge` in $O(\log n)$ time, where n is the total number of keys in priority queues H_1 and H_2. Unfortunately, it does not seem to be possible to do this with the standard binary heap data structure because of its highly rigid array-based structure.

Leftist Heaps: We introduce a new data structure called a *leftist heap*, which supports the operations `insert(x)`, `extract-min()`, and `merge(H1,H2)`. Like the binary heap, it is a binary tree, but it is stored as a standard binary tree with left and right child pointers. In order to support operations in $O(\log n)$ time, we want the tree's height to be $O(\log n)$. Our data structure will have a weaker property, which is where the term "leftist" comes from.

Leftist Heap Property: Define the *null path length*, denoted `npl(v)`, of any node v to be the length of the shortest path to reach a node with a `null` child pointer. The value of `npl(v)` can be defined recursively as follows (see Fig. 5).

$$\texttt{npl(v)} = \begin{cases} \texttt{-1} & \text{if } \texttt{v} = \texttt{null}, \\ \texttt{1 + min(npl(v.left), npl(v.right))} & \text{otherwise.} \end{cases}$$

Note that the npl value of any (non-null) node is at least zero. The npl of a node is very different from its height. Trees with very large heights can have very small npl value.

Leftist: A node `v` is *leftist* if `npl(v.left)` $\geq$ `npl(v.right)`.

Leftist heap: Is a binary tree whose keys are heap ordered (parent key is less than or equal to child key) and whose nodes' npl values all satisfy the leftist property.

For example, the two trees shown in Fig. 5 are both heap-ordered, but the tree in part (a) is not leftist because node 6's left child has a smaller npl value than its right child. Swapping these two children as shown in (b) yields a valid leftist heap.

Note that any tree that does not satisfy leftist property can always be made to do so by swapping left and right subtrees at any nodes that violate the leftist property. Observe that this does not affect the heap-order property. The key to the efficiency of leftist heap operations is that there exists a short ($O(\log n)$ length) path in every leftist heap, namely the rightmost path. We prove the following lemma, which implies that the rightmost path in a leftist tree cannot be of length greater than $O(\log n)$.

Lemma: A leftist binary tree with $r \geq 1$ nodes on its rightmost path has has at least $2^r - 1$ nodes.

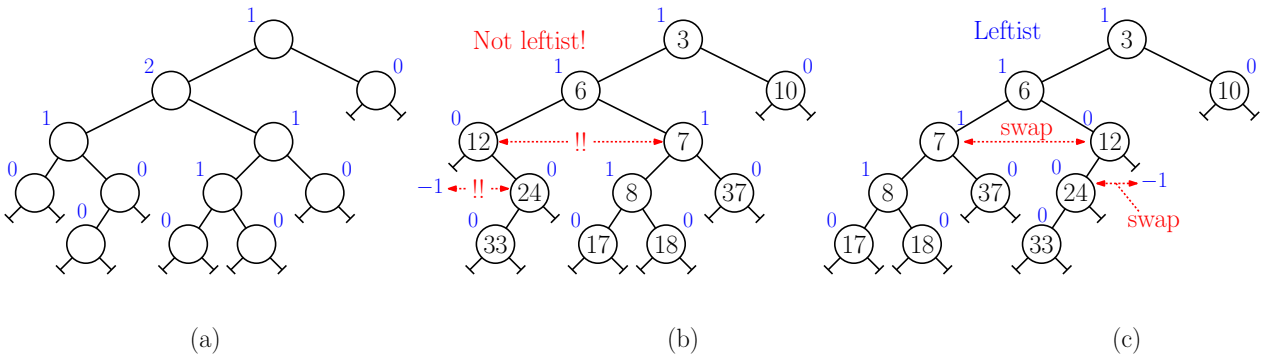

Fig. 5: Null path lengths and the leftist property.

Proof: The proof is by induction on the size of the rightmost path. Before beginning the proof, we begin with two observations, which are easy to see: (1) the shortest path in any leftist heap is the rightmost path in the heap, and (2) any subtree of a leftist heap is a leftist heap. For the basis case, if there is only one node on the rightmost path, then the tree has at least one node. Since $1 \geq 2^1 - 1$, the basis case is satisfied.

For the induction step, let us suppose that the lemma is true for any leftist heap with strictly fewer than r nodes on its rightmost path, and we will prove it for a binary tree with exactly r nodes on its rightmost path. Remove the root of the tree, resulting in two subtrees. The right subtree has exactly $r - 1$ nodes on its rightmost path (since we have eliminated only the root), and the left subtree must have a at least $r - 1$ nodes on its rightmost path (since otherwise the rightmost path in the original tree would not be the shortest, violating (1)). Thus, by applying the induction hypothesis, it follows that the right and left subtrees have at least $2^{r-1} - 1$ nodes each, and summing them, together with the root node we get a total of at least

$$2(2^{r-1} - 1) + 1 \;=\; 2^r - 1$$

nodes in the entire tree.

Corollary: The rightmost path of a leftist binary tree with n nodes has length $O(\log n)$.

Proof: By the above lemma, $n \geq 2^r - 1$. Thus, $r \leq \lg(n + 1) = O(\log n)$.

Class Structure: Before discussing how merging is performed in leftist heaps, it would be good to give an overview of the class structure (see the code block below). The class `LeftistHeap` is generic, and its declaration is templated by the key type `Key`. Because we need to compare keys, we inform the compiler that the key type must implement the `Comparable` interface.

The class consists of three major components, a class declaration for the node type, the private data, and the various public and private methods. The node, called `LHNode` stores the key, the child pointers, and the npl value for this node. `LeftistHeap` has one piece of private date, namely the root of the tree, and the class constructor just sets the root to `null`, thus generating an empty tree. This is followed by the other public and private members. These include the public methods `insert`, `extractMin`, and `mergeWith`. We have omitted all the details, including addition private helper methods.

The public merge method that performs the merger is called `mergeWith`, and it merges this heap with another heap `H2`. It invokes a recursive helper function (given below) and updates

Leftist Heap Class Structure

```
public class LeftistHeap<Key extends Comparable<Key>> {

    private class LHNode {                  // a node in the tree
        Key key;                            // key
        LHNode left, right;                 // children
        int npl;                            // null path length
    }

    private LHNode root;                    // root of the tree

    public LeftistHeap() { root = null; }   // constructor
    public void insert(Key x) { ... }       // insert
    public Key extractMin() { ... }         // extract-min
    public void mergeWith(LeftistHeap<Key> H2) { ... } // merge with H2
}
```

the root to point to the resulting tree. Note that `H2` is destroyed in the process, and we signal this by setting its root to `null`.

```
public void mergeWith(LeftistHeap H2) {     // merge this heap with H2
    root = merge(this.root, H2.root)        // invoke the helper and update root
    H2.root = null                          // H2 is destroyed in the process
}
```

Merging Leftist Heaps: All that remains, is to show how to implement the helper function `merge(u, v)`. This takes two nodes `u` and `v`, one from each of the heaps, merges the subtrees rooted at these two nodes, and returns a pointer to the root node of the merged tree.

The formal description of the procedure is recursive. However it is somewhat easier to understand in its nonrecursive form, which operates in two separate phases. In the first phase, we walk down the rightmost paths of both subtrees. By the heap ordering, the keys along each of these paths form an increasing sequence. We merge these paths into a single sorted path by selecting the one with the smaller key value (see Fig. 6).

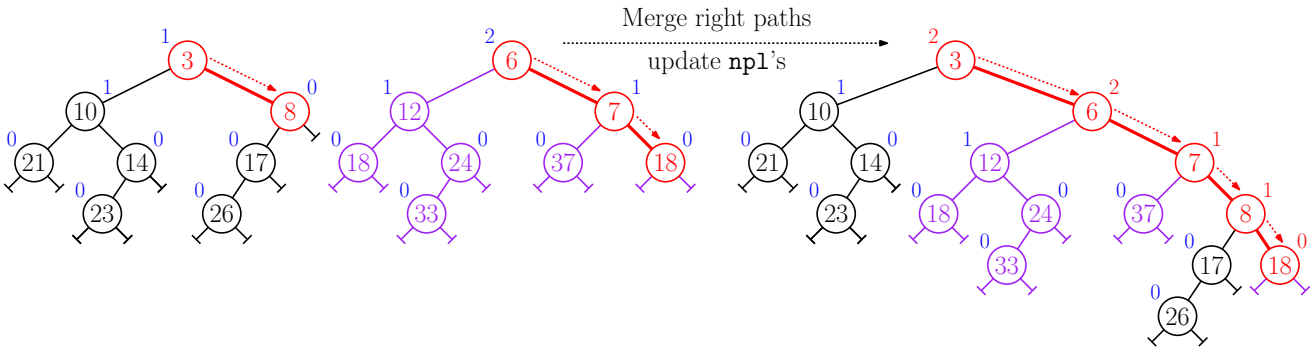

Fig. 6: Merging leftist heaps (Phase-1): Merge rightmost paths by key.

Each node of the tree contains an additional field `npl`, which stores the npl value for this node. After merging the right paths, we update the npl values. After the merger is done,

the leftist property may be violated along this rightmost path. To remedy this, we perform a second phase, which swaps left and right children to restore the leftist property. (Recall that this swapping preserves the heap ordering.) This is illustrated in Fig. 7.

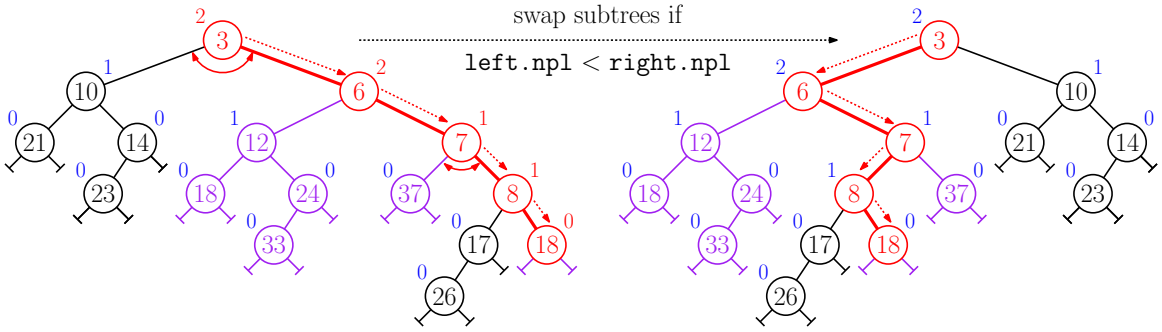

Fig. 7: Merging leftist heaps (Phase-2): Swap left-right to restore leftist property.

Merge two leftist heaps

```
LHNode merge(LHNode u, LHNode v) {          // recursive helper function
    if (u == null) return v                 // if one is empty, return the other
    if (v == null) return u
    if (u.key > v.key) swap(u, v)           // swap so that u has smaller key
    if (u.left == null) u.left = v          // u must be a leaf in this case
    else {                                  // merge v on right and swap if needed
        u.right = merge(u.right, v)         // recursively merge u's right subtree
        if (u.left.npl < u.right.npl) {     // fail the leftist property?
            swap(u.left, u.right)           // swap to restore leftist
        }
        u.npl = u.right.npl + 1             // update npl value
    }
    return u                                // return the root of final tree
}
```

The complete code is given in the code block. It is expressed in recursive form, and both phases are combined into a single phase. First, it deals with the trivial cases if either `u` or `v` is empty (`null`), in which case it simply returns the other tree as the answer. It then swaps `u` and `v` so that `u` contains the smaller key. If `u` has no left child (which by leftism implies it has no right child), then we attach the entire subtree `v` as the left child of `u`. (Leftism demands that we put it on the left side.) We then recursively merge the remainder of `u`'s right subtree with `v`'s tree. After this returns, we swap the left and right subtrees if needed to satisfy the leftist property at `u`. Finally, we update `u`'s npl value and return `u` as the final result. (I would recommend tracing it out on the above example from Fig. 6 to see that it produces the same result as in Fig. 7.)

For the analysis, observe that because the recursive algorithm spends $O(1)$ time for each node on the rightmost path of either u or v, the total running time is $O(\log n)$, where n is the total number of nodes in both heaps.