**CS2040S: Data Structures and Algorithms**

# Recitation 9

sorted Array
Cost of creating
Heap is O(n) NOT O(n log n)

*Goals:*

- Apply ideas from Union-Find
- Explore more heap variants and their properties
- Reinforce algorithm design principles

## Problem 1. A whole lot of work to do....

Suppose you have a huge amount of work to do. To keep track of the $n$ tasks, you have a big array $W$ where $W[i] = \texttt{0}$ means that task $i$ has not yet been completed, and $W[i] = \texttt{1}$ implies that task $i$ has been completed.

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

**Figure 1:** An example of $W$.

In order to help coordinate getting all the work done, you want to implement a data structure that implements the following three operations:

| **Operation** | **Behaviour** |
|---|---|
| `lookup(i)` | Returns the value $W[i]$. |
| `mark(i)` | Marks task $i$ as completed, i.e., sets $W[i] \leftarrow 1$. |
| `nextTask(i)` | Returns the next task from $i$ onwards that is not yet completed, i.e., the next index $j \geq i$ where $W[j] = \texttt{0}$. |

For example, for the array above (Figure 1), `lookup(2)` returns 1, `mark(4)` would change the zero to a one, and `nextTask(6)` would return 10.

The simplest solution, of course, is just to use the array. In that case, it is easy to implement the `lookup` and `mark` operations in $O(1)$ time, but the `nextTask` operation may take $\Omega(n)$ time.

The goal in this problem is to develop data structures that provide different trade-offs in performance. For all versions, we want `lookup` to complete in $O(1)$ time.

**Problem 1.a.** Design a data structure where the `mark` and `nextTask` operations complete in $O(\log n)$ time, worst-case.

**Problem 1.b.** Design a data structure where the `mark` operation runs in $O(\log n)$ *amortized* time and `nextTask` operation runs in $O(1)$ time, worst-case.

**Problem 1.c.** Design a data structure where the `mark` operation runs in worst-case $O(1)$ time and `nextTask` operation runs in $O(\alpha(n))$ *amortized* time (where $\alpha(n)$ refers to the time complexity of the inverse Ackermann function, i.e., the amortized time for executing $n$ operations on a union-find data structure).

## Problem 2. Communist Data Structures

In this problem we will build a new type of mergeable Max-Heap. It is often useful to be able to merge data structures. For example, they can be used to build divide-and-conquer algorithms, they can be used more easily in augmenting trees, etc.

Let us first define the following terms:

| Term | Definition |
|---|---|
| right spine | The sequence of nodes traversed in a tree if you start at a node and always go right until you find a node with no right child (which may not be a leaf). |
| $u$`.rightRank` | The number of nodes along the right spine of node $u$. |
| LEFTIST property | The property a tree satisfies if, for every node, `rightRank(`$L$`) >= rightRank(`$R$`)` where $L$ and $R$ are the left and right child respectively. The `rightRank` for a non-existent child is treated to be zero. |
| LEFTIST (Max) HEAP | A tree that satisfies both the (Max) Heap order property and the LEFTIST property. |

Below is the ADT specification of a LEFTIST HEAP.

| Operation | Behaviour |
|---|---|
| `insert(u)` | Insert node `u` into the heap. |
| `merge(t2)` | Returns the tree as a result of merging the heap with `t2`. |
| `extractMax()` | Removes the node with the maximum key from the heap and return it. |

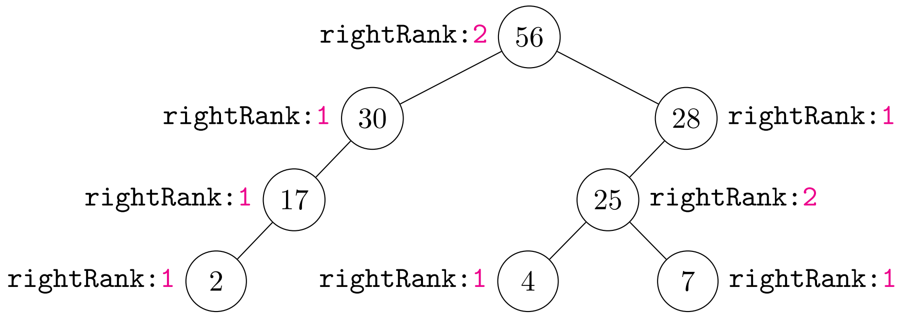


**Figure 2:** Example of a LEFTIST HEAP, where the `rightRank` for each node is labelled next to it.

**Problem 2.a.** What is the maximum height/depth of a LEFTIST HEAP?

**Problem 2.b.** What is the maximum `rightRank` of a LEFTIST HEAP containing $n$ nodes?

**Problem 2.c.** How would you implement the `insert` operation for a LEFTIST HEAP?

Where would be a good place to insert the new node? What is the maximum cost?

Note that your strategy must work on *any valid* LEFTIST HEAP. i.e. it need not be one built sequentially from the insertion algorithm you propose.

*Hints:* Think recursively! Where is an efficient place to insert it? Realise also you can always swap the left and right subtrees of a node if it does not satisfy the LEFTIST property.

**Problem 2.d.** Now come up with an algorithm to implement the `merge` operation. What is the running time?

*Hint:* Apply the same idea from part b.

**Problem 2.e.** How would you implement `extractMax` in a LEFTIST HEAP?

**Problem 2.f.** Now suppose we extend the LEFTIST HEAP ADT with the following additional operations:

| **Operation** | **Behaviour** |
|---|---|
| `delete(`$u$`)` | Delete the node $u$ from the heap. |
| `updateKey(`$u$`, `$k$`)` | Update the key of node $u$ in the heap to $k$. |

How would you implement them efficiently? Note that for both these operations, you do not have to search for node $u$ since it is provided as an argument.

*Hint:* You should make use of what you have already come up with from earlier parts.