Topic: Heaps and priority queues

• The priority-queue problem
• The heap solution
• Binary trees and complete binary trees
• Running time of heap operations
• Array implementation of heap
• Operations: Heap-insert, Heapify, Heap-extract, Heapsort

The priority-queue problem

Example:
• We wish to store 8 print jobs while they wait for printer time
• We wish to assign each job a priority, say, (2, 1, 4, 1, 3, 2, 3, 5)
• We will want them to be sent to the printer in order of priority
• This is a job for a priority queue
Priority queue solution

- A priority queue is a collection with restricted access
- Each item in priority queue has a key value denoting its relative priority
- Operations: Insert, Extract
  - Extract retrieves and deletes the item of highest priority
  - Minimum queue treats low-valued key as high-priority; maximum queue prioritizes high-valued key

Sorting with a priority queue

PQ-Sort \((A)\)

Initialize \((PQ)\)

For \(i \leftarrow 1\) to \(\text{size}(A)\)
  - Insert \((A_i, PQ)\)

For \(i \leftarrow 1\) to \(\text{size}(A)\)
  - \(A_i \leftarrow \text{Extract}(PQ)\)

- Precondition: \(A\) is any sequential or random-accessible collection
- Postcondition: \(A\) is sorted ascending
- Complexity: \(n \times T_{\text{Insert}}(n) + n \times T_{\text{Extract}}(n)\)
A simple, inefficient priority queue

- Store jobs in array in chronological order, with their priorities

<table>
<thead>
<tr>
<th>Job</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- Repeatedly retrieve, run, and delete first job found with the top priority (Extract)

- Complexity of Extract: $O(\_\_\_)$

  of Insert: $O(\_\_\_)$

Trading Insert’s efficiency for Extract’s

- Store jobs in order of priority:

<table>
<thead>
<tr>
<th>Job</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>F</th>
<th>E</th>
<th>G</th>
<th>C</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

(Cronological order here is ABCD…)

- Top-priority job is last in array

- Complexity of Extract: $O(\_\_\_)$

  of Insert: $O(\_\_\_)$
Discussion problems

1. What are complexities of priority queues based on linked lists:
   - w/unsorted list \( O(\_\_) \quad O(\_\_) \)
   - w/ascending list \( O(\_\_) \quad O(\_\_) \)

2. Could a doubly linked list improve a priority queue’s efficiency?

3. Can the sum of the complexities of \( \text{Insert} \) and \( \text{Extract} \) be improved?

4. How does priority queue differ from queue data structure?

Priority-queue design problem

- Can a priority queue be built that attains running times
  - Better than \( O(n) \) for \( \text{Insert} \)
  and
  - Better than \( O(n) \) for \( \text{Extract} \)
- Can it be done using an array?
A more efficient priority queue

- In a minimum heap, the lowest-valued item is always at the front of queue (top of heap)

- A maximum heap reverses order rule

- Claim: this feature assures fast insertion and fast retrieval of the top-priority item

A tree is a branching structure

- In a tree, a node may be linked to more than one other node

- In a binary tree, each node may have up to two child nodes:
  - a left
  - a right

- a subtree of the larger tree

- root

- a’s children

- leaves
Parent and child nodes in a binary tree

• each node is adjacent to, and parent of, 0, 1 or 2 nodes that are its children
• each node except the root is adjacent to exactly one parent node

A binary tree

Not a binary tree

A heap is a kind of tree

• The heap property (for minimum heaps):
The priority value stored in each node is less than or equal to the value stored in either child
• Heap property applies to all nodes of a heap
• The heap property ensures that highest priority node is in root at all times
A heap is a kind of complete binary tree

- A complete binary tree has all levels full except possibly the bottom one, which is filled from the left
- Which are complete binary trees?

Array implementation of complete binary tree

- Root is $A[1]$
- Parent of $A[i]$ is $A\lfloor i/2 \rfloor$
- If $n$ is size, height is $\lceil \log_2 n \rceil$
Node depth and tree height

- **depth** or **level** of a node in a binary tree: length (number of edges) of the (unique) path from the root to the node
- **height** of tree: the number of nodes in the longest path from root to a node
- A **level** in a tree is the set of nodes of a given depth

A level in a tree is shown below:

```
Level

0

1

2

3

Tree height: 4
```

Height of a complete binary tree with \( n \) nodes is \( O(\log_2 n) \)

- ...because size \( n \) approximately doubles with each level added
- Or, \( 2^{\text{height} - 1} \leq n \leq 2^{\text{height}} - 1 \)
Array implementation of heap

- Uses array implementation of complete binary tree
- Pointers are not used
- Left-right relationships not significant
- Running time for access to
  - root, from root? ______
  - a node’s parent, given node’s subscript? ______
  - farthest node from root? ______
  - root, from a leaf? ______
  - leaf, from root: ______

A maximum heap

- Revised heap property: for all $i$ up to array size, $A[i] \leq A[i/2]$
The **Heapify** operation

- **Heapify**(A, i) applies to one node, with subscript i, of a heap stored in array A
- It assumes that subtrees with roots **Left-child**(i) and **Right-child**(i) already have the heap property
- Node A[i] may meet or violate heap property
- **Heapify**(A, i) causes the subtree with root subscript i to have the heap property

**Intuition for Heapify**

- The algorithm drags the value at a selected node down to its proper level where heap property applies to it
- It does this by recursively exchanging the value in the root node with the higher-priority child of its two children
- **Complexity**: Depth of complete binary tree with n nodes, i.e., O(____)
Heapify($A, i$) for minimum heap

$L \leftarrow \text{Left}(i)$
$R \leftarrow \text{Right}(i)$

if $L \leq \text{Heap-size}(A)$ and $A[L] < A[i]$  

$\text{smallest} \leftarrow L$

else

$\text{smallest} \leftarrow i$

if $R \leq \text{Heap-size}(A)$ and $A[R] < A[\text{smallest}]$  

$\text{smallest} \leftarrow R$

if $\text{smallest} \neq i$

exchange $A[i]$ with $A[\text{smallest}]$

Heapify($A, \text{smallest}$)

---

To convert an array to a heap

*Overview:* Heapify at every non-leaf node, starting at the bottom

**Build-heap($A$)**

$\text{Heap-size}[A] \leftarrow \text{length}[A]$

for $i \leftarrow \lfloor \text{length}[A] / 2 \rfloor$ down to 1

Heapify($A, i$)

*Use version for min or max heap*

*Because leaves don’t need to be heapified*

**Running time:** $O(\quad)$
Extract value from heap

- Returns minimum value from min-heap
- Deletes that value from heap

**Extract-min (A)**

If $\text{Heap-size}(A) < 1$

- Display message “Empty heap”

$min \leftarrow A[1]$

$A[1] \leftarrow A[\text{Heap-size}(A)]$

$\text{Heap-size}(A) \leftarrow \text{Heap-size}(A) - 1$

Heapify(A,1)

Return $min$

[HERE GOES A SERIES OF SLIDES FROM TOMASSIA]
Intuition for *Heap-insert*

- Algorithm starts at a new leaf node
- Loop ascends a path toward root
- New node’s value is in effect swapped upward until it reaches proper level

**Insertion into a min-heap**

**Heap-insert** \((A, key)\)

\[
\text{Heap-size}(A) \leftarrow \text{Heap-size}(A) + 1 \\
i \leftarrow \text{Heap-size}(A) \\
\text{while } i > 1 \text{ and } A[\text{Parent}(i)] > key \\
A[i] \leftarrow A[\text{Parent}(i)] \\
i \leftarrow \text{Parent}(i) \\
A[i] \leftarrow \text{key}
\]

Running time: \(O(\_\_\_\_\_\_)\)
To sort an array using a max-heap

\begin{verbatim}
Heapsort(A)
    Build-heap(A)
    for i ← length(A) downto 2
        Heap-size(A) ← Heap-size(A) − 1
        Heapify(A,1)
\end{verbatim}

- In plain language...
  Repeatedly extract the highest value from heap and put it at front of a growing sorted array just after the heap
- Complexity: O( )
Summary of heap implementation of priority queue

- Uses array to manage memory
- Insertion, $Extract-min$ are logarithmic-time
- Enables $n \log n$ sorting time
- Further study: mergeable heaps, e.g., binomial heap