Array Implementation of Binary Trees

Each node $v$ is stored at index $i$ defined as follows:
- If $v$ is the root, $i = 1$
- The left child of $v$ is in position $2i$
- The right child of $v$ is in position $2i + 1$
- The parent of $v$ is in position $???
Space Analysis of Array Implementation

- $n$: number of nodes of binary tree $T$
- $p_M$: index of the rightmost leaf of the corresponding **full** binary tree (or size of the full tree)
- $N$: size of the array needed for storing $T$; $N = p_M + 1$

Best-case scenario: balanced, full binary tree $p_M = n$

Worst case scenario: unbalanced tree

- Height $h = n - 1$
- Size of the corresponding full tree:
  \[ p_M = 2^{h+1} - 1 = 2^n - 1 \]
- $N = 2^n$

Space usage: $O(2^n)$

Array versus Linked List

**Linked lists**
- Slower operations due to pointer manipulations
- Use less space if the tree is unbalanced
- Rotation (restructuring) code is simple

**Arrays**
- Faster operations
- Use less space if the tree is balanced (no pointers)
- Rotation (restructuring) code is complex
Priority Queues

- Data structure supporting the following operations:
  - insert (equivalent to enqueue)
  - deleteMin (or deleteMax) (equivalent to dequeue)
  - Other operations (optional)

- Applications:
  - Emergency room waiting list
  - Routing priority at routers in a network
  - Printing job scheduling

Simple Implementations of PQs

- Unsorted linked list
  - insertion $O(n)$
  - deleteMin $O(n)$
- Sorted linked list
  - insertion $O(n)$
  - deleteMin $O(n)$
- Red black trees
  - insertion $O(\log n)$
  - deleteMin $O(\log n)$
- Unsorted array
  - insertion $O(n)$
  - deleteMin $O(n)$
- Sorted array
  - insertion $O(n)$
  - deleteMin $O(n)$
- A data structure more efficient for PQs is heaps.
Heaps

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-Order: for every internal node \( v \) other than the root, \( \text{key}(v) \geq \text{key}(\text{parent}(v)) \)
  - Complete Binary Tree: let \( h \) be the height of the heap
    - for \( i = 0, \ldots, h - 1 \), there are \( 2^i \) nodes at depth \( i \)
    - at depth \( h \), the nodes are filled from left to right

Examples that are not heaps

- bottom level is not left-filled
- key(parent) > key(child)
**Height of a Heap**

- Theorem: A heap storing $n$ keys has height $O(\log n)$
  
  **Proof:** (we apply the complete binary tree property)
  
  - Let $h$ be the height of a heap storing $n$ keys.
  - Since there are $2^i$ keys at depth $i = 0, \ldots, h - 1$ and at least one key at depth $h$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$

<table>
<thead>
<tr>
<th>depth</th>
<th>keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$h-1$</td>
<td>$2^{h-1}$</td>
</tr>
<tr>
<td>$h$</td>
<td>1</td>
</tr>
</tbody>
</table>

**Some Notes**

- Given a complete binary tree of height $h$ and size $n$, $2^h \leq n \leq 2^{h+1} - 1$

- The definition we just discussed is for a *min* heap. Analogously, we can declare a *max* heap if we need to implement `deleteMax` operation instead of `deleteMin`.

- Which data structure is better for implementing heaps, arrays or linked structure?
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node

Insertion into a Heap

- Method insert of the priority queue ADT corresponds to the insertion of a key $k$ to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node $z$ (the new last node)
  - Store $k$ at $z$
  - Restore the heap-order property (discussed next)
Upheap Percolation

- After the insertion of a new key $k$, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping $k$ along an upward path from the insertion node
- Upheap terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

Removal from a Heap

- Method `deleteMin` of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node $w$
  - Remove $w$
  - Restore the heap-order property (discussed next)
Downheap Percolation

- After replacing the root key with the key $k$ of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key $k$ along a downward path from the root
- Upheap terminates when key $k$ reaches a leaf or a node whose children have keys greater than or equal to $k$
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

Heap Sort

- Consider a priority queue with $n$ items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and deleteMin take $O(\log n)$ time
  - methods size, isEmpty, and findMin take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of $n$ elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort
Heap Sort Using a Heap

- Input: array A with n elements to be sorted
- Create a heap T
- Sorting code
  
  ```
  for (i = 0; i < n)
    T.insert(A[i]);
  for (i = 0; i < n)
    A[i] = T.deleteMin();
  ```

Analysis of Heap Sort

- Stirling’s approximation: 
  
  \[ n! \approx n^n e^{-n} \sqrt{2\pi n} \]

- Insertions
  
  \[ \log 1 + \log 2 + \ldots + \log n = \log(n!) = O(n \log n) \]

- Deletions
  
  \[ \log 1 + \log 2 + \ldots + \log n = \log(n!) = O(n \log n) \]

- Total = \( O(n \log n) \)
In-place Heap Sort

- The heap sort algorithm we just discussed requires a temporary array (the heap).
- In-place heap sort uses only one array, which is the original array storing the inputs.
- Needs to build a heap (in-place) from the set of input values ⇒ buildHeap procedure

buildHeap

- Input: a non-heap binary tree stored in an array
- Output: a heap stored in the same array
- We can construct a heap storing \( n \) given keys in using a bottom-up construction with \( \log n \) phases
- In phase \( i \), pairs of heaps with \( 2^i - 1 \) keys are merged into heaps with \( 2^{i+1} - 1 \) keys
Examples

- See demo with max heaps at www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html

Analysis of $\text{buildHeap}$

- Bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than $n$ successive insertions and speeds up the first phase of heap-sort
Analysis of buildHeap (cont’d)

- Theorem: For the perfect binary tree of height $h$ containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of all the nodes is $2^{h+1} - 1 - (h + 1)$

- buildHeap thus runs in $O(n)$ time

In-place Heap Sort (Section 7.5)

- The first step is to build a max heap using buildHeap
- Call deleteMax to remove the max item (the root). The heap size is reduced by one. The last entry of the heap is now empty. Store the item just removed into that location (copyMax).
- Repeat deleteMax and copyMax until the heap is empty.
- Examples: will be shown in class
- Demo-animation
  - www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heapSort.html
  - www2.hawaii.edu/~copley/665/HSApplet.html
  - www-cse.uta.edu/~holder/courses/cse2320/lectures/applets/sort1/heapsort.html
More Heap Operations

- `decreaseKey(p, d)`
  - `T[p] = T[p] − d`, then percolate up
  - Example: system admin boosts the priority of their jobs
- `increaseKey(p, d)`
  - `T[p] = T[p] + d`, then percolate down
  - Example: penalizing misbehaved processes
- `delete(p)`
  - Perform `decreaseKey(p, ∞)` then `deleteMin()`
  - Example: removing a print job from the PQ

Note: searching for index `p` takes O(n) time in the worst case, but we expect not to use the above methods very often.

Next Topics …

- Randomization and Skip Lists
- Graphs

 Heap animation:
  www.student.seas.gwu.edu/~idsv/idsv.html