Lecture 10/21/16

Lecturer: Xiaodi Wu

Reading: Chapter 2.4, [CLRS] Chap 6
Total Order & Comparator

**Total Order**

$\leq$, defined on every pair of elements, such that

- **Reflexive**: $k \leq k$.
- **Anti-symmetric**: $k_1 \leq k_2$ and $k_2 \leq k_1 \Rightarrow k_1 = k_2$.
- **Transitive**: $k_1 \leq k_2$ and $k_2 \leq k_3 \Rightarrow k_1 \leq k_3$. 

**Comparators**

A comparator is an object that defines a total order on elements in the following way:

- `isLess(a,b)`
- `isLessOrEqualTo(a,b)`
- `isEqualTo(a,b)`
- `isGreater(a,b)`
- `isGreaterOrEqualTo(a,b)`
Total Order & Comparator

Total Order
≤, defined on every pair of elements, such that
- **Reflexive**: $k \leq k$.
- **Anti-symmetric**: $k_1 \leq k_2$ and $k_2 \leq k_1 \Rightarrow k_1 = k_2$.
- **Transitive**: $k_1 \leq k_2$ and $k_2 \leq k_3 \Rightarrow k_1 \leq k_3$.

Comparaters
A comparater is an object that defines a total order on elements in the following way:
- isLess(a,b), isLessOrEqualTo(a,b)
- isEqualTo(a,b)
- isGreater(a,b), isGreaterOrEqualTo(a,b)
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by **keys**. Each element is associated with a **key**.
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by **keys**. Each element is associated with a **key**.

- **insertItem(k, e)**: insert an element **e** with key **k** into PQ.
- **removeMin()**: Return and remove from PQ an element with the **smallest** key.
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.

- \texttt{insertItem}(k, e): insert an element \( e \) with key \( k \) into PQ.
- \texttt{removeMin}(): Return and remove from PQ an element with the smallest key.

Simple Implementation on top of Queues

- Store Keys.
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.

- insertItem\((k, e)\): insert an element e with key k into PQ.
- removeMin(): Return and remove from PQ an element with the smallest key.

Simple Implementation on top of Queues

- Store Keys.
- insertItem\((k, e)\): \(O(1)\), removeMin(): \(O(n)\). How?
Priority Queue (PQ)

Similar to queues, however, insertion and removal principle determined by keys. Each element is associated with a key.

- \textbf{insertItem}(k, e): insert an element e with key k into PQ.
- \textbf{removeMin}(): Return and remove from PQ an element with the smallest key.

Simple Implementation on top of Queues

- Store Keys.
- \textbf{insertItem}(k, e): \(O(1)\), \textbf{removeMin}(): \(O(n)\). How?
- \textbf{insertItem}(k, e): \(O(n)\), \textbf{removeMin}(): \(O(1)\). How?
Algorithm PQ-sort($C, P$)
Input: an $n$-element sequence $C$, a priority queue $P$.
Output: the sequence $C$ sorted by the total order relation.

while ! $C$.isEmpty() do
    $e \leftarrow C$.removeFirst()
    $P$.insertItem($e$, $e$).
end while

while ! $P$.isEmpty() do
    $e \leftarrow P$.removeMin().
    $C$.insertLast($e$).
end while
Algorithm PQ-sort\((C, P)\)
Input: an \(n\)-element sequence \(C\), a priority queue \(P\).
Output: the sequence \(C\) sorted by the total order relation.
\[
\text{while } \neg C.\text{isEmpty()} \text{ do}
\]
\[
e \leftarrow C.\text{removeFirst}()
\]
\[
P.\text{insertItem}(e, e).
\]
\[
\text{end while}
\]
\[
\text{while } \neg P.\text{isEmpty()} \text{ do}
\]
\[
e \leftarrow P.\text{removeMin}().
\]
\[
C.\text{insertLast}(e).
\]
\[
\text{end while}
\]

Correctness?
PQ-based Sorting: Simple Implementation

- `insertItem(k, e)`: $O(1)$, `removeMin()`: $O(n)$. Total running time $O(n^2)$. Also known as "selection-sort".

- `insertItem(k, e)`: $O(n)$, `removeMin()`: $O(1)$. Total running time $O(n^2)$. Also known as "insertion-sort".

- `insertItem(k, e)`: $O(\log n)$, `removeMin()`: $O(\log n)$. Total running time $O(n \log n)$. Also known as "heap-sort".

Optimal running time? Yes for comparison-based sorting.

Week 9, 10!
PQ-based Sorting: Simple Implementation

- `insertItem(k, e)`: $O(1)$, `removeMin()`: $O(n)$. Total running time $O(n^2)$. Also known as "selection-sort".
- `insertItem(k, e)`: $O(n)$, `removeMin()`: $O(1)$. Total running time $O(n^2)$. Also known as "insertion-sort".

Improvement on efficiency?

- `insertItem(k, e)`: $O(\log n)$, `removeMin()`: $O(\log n)$. Total running time $O(n \log n)$. Also known as "heap-sort".

Optimal running time? Yes for comparison-based sorting.

Week 9, 10!
PQ-based Sorting: Simple Implementation

- **insertItem**($k, e$): $O(1)$, **removeMin()**: $O(n)$. Total running time $O(n^2)$. Also known as ”selection-sort”.
- **insertItem**($k, e$): $O(n)$, **removeMin()**: $O(1)$. Total running time $O(n^2)$. Also known as ”insertion-sort”.

**Improvement on efficiency?**

- **insertItem**($k, e$): $O(\log n)$, **removeMin()**: $O(\log n)$. Total running time $O(n \log n)$. Also known as ”heap-sort”.
PQ-based Sorting: Simple Implementation

- `insertItem(k, e)`: $O(1)$, `removeMin()`: $O(n)$.
  Total running time $O(n^2)$. Also known as "selection-sort".
- `insertItem(k, e)`: $O(n)$, `removeMin()`: $O(1)$.
  Total running time $O(n^2)$. Also known as "insertion-sort".

Improvement on efficiency?

- `insertItem(k, e)`: $O(\log n)$, `removeMin()`: $O(\log n)$.
  Total running time $O(n \log n)$. Also known as "heap-sort".
- Optimal running time? Yes for comparison-based sorting. Week 9, 10!
How to achieve $O(\log n)$ for both insertion and removal?

Heap
Instead of storing elements in sequences, store in the internal nodes of complete binary trees satisfying the Heap-Order Property.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in the internal nodes of complete binary trees satisfying the Heap-Order Property.

- External Nodes as ”place-holders”.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in the **internal nodes** of **complete binary trees** satisfying the **Heap-Order Property**.

- External Nodes as "place-holders".
- Heap-Order Property: for every node $v$ other than the root, its key $\geq$ the key of its parent.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in the **internal nodes** of **complete binary trees** satisfying the **Heap-Order Property**.

- External Nodes as ”place-holders”.
- Heap-Order Property: for every node $v$ other than the root, its key $\geq$ the key of its parent.
- Complete Binary Trees: binary tree with height $h$ and maximum number of nodes in all levels $0, \cdots, h - 1$. In level $h - 1$, the internal nodes are to the left of the external nodes.
How to achieve $O(\log n)$ for both insertion and removal?

Heap

Instead of storing elements in sequences, store in the **internal nodes** of **complete binary trees** satisfying the **Heap-Order Property**.

- **External Nodes** as ”place-holders”.
- **Heap-Order Property**: for every node $v$ other than the root, its key $\geq$ the key of its parent.
- **Complete Binary Trees**: binary tree with height $h$ and maximum number of nodes in all levels $0, \cdots, h - 1$. In level $h - 1$, the internal nodes are to the left of the external nodes.
- **Last Node**: as the rightmost internal node on level $h - 1$. 
Heap Example: only keys
Heap Property

Theorem (2.10)

A heap $T$ storing $n$ keys has height $h = \lceil \log(n + 1) \rceil$
Heap Property

Theorem (2.10)
A heap $T$ storing $n$ keys has height $h = \lceil \log(n + 1) \rceil$

Proof.
- The total number of internal nodes at least is $2^{h-1}$. 

Remark: if updates $\sim$ height $h$, then $O(\log(n+1))$. 
Heap Property

Theorem (2.10)

A heap $T$ storing $n$ keys has height $h = \lceil \log(n + 1) \rceil$

Proof.

- The total number of internal nodes at least is $2^{h-1}$.
- The total number of internal nodes is at most $2^h - 1$.
Theorem (2.10)
A heap $T$ storing $n$ keys has height $h = \lceil \log(n + 1) \rceil$

Proof.

- The total number of internal nodes at least is $2^{h-1}$.
- The total number of internal nodes is at most $2^h - 1$.
- Thus, $\log(n + 1) \leq h \leq \log(n) + 1$ and $h$ is an integer.
Heap Property

Theorem (2.10)

A heap \( T \) storing \( n \) keys has height \( h = \lceil \log(n + 1) \rceil \)

Proof.

- The total number of internal nodes at least is \( 2^{h-1} \).
- The total number of internal nodes is at most \( 2^h - 1 \).
- Thus, \( \log(n + 1) \leq h \leq \log(n) + 1 \) and \( h \) is an integer.

Remark: if updates \( \sim \) height \( h \), then \( O(\log(n)) \).
Vector-based Implementation

Binary-Tree

- \( p(v) \): the rank of \( v \) stored in array \( A \) of size \( N \).
- If \( v \) is the root, then \( p(v) = 1 \).
- If \( v \) is the left child of \( u \), then \( p(v) = 2p(u) \).
- If \( v \) is the right child of \( u \), then \( p(v) = 2p(u) + 1 \).

Convention: \( p(v) = \) index + 1 in storage!

Application to heaps

- The last node of a heap of \( n \) keys is indexed \( n \) in the array.
- The first empty external node is then indexed \( n + 1 \).
- Don't need to store external nodes explicitly.
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$.
- **Convention:** $p(v) = \text{index} + 1$ in storage!

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
- The first empty external node is then indexed $n + 1$.
- Don't need to store external nodes explicitly.
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$.
- **Convention**: $p(v) = \text{index} + 1$ in storage!

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$.
- **Convention:** $p(v) = \text{index} + 1$ in storage!

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
- The first empty external node is then indexed $n + 1$. 
Vector-based Implementation

Binary-Tree

- $p(v)$: the rank of $v$ stored in array $A$ of size $N$.
- If $v$ is the root, then $p(v) = 1$.
- If $v$ is the left child of $u$, then $p(v) = 2p(u)$.
- If $v$ is the right child of $u$, then $p(v) = 2p(u) + 1$.
- **Convention**: $p(v) = \text{index} + 1$ in storage!

Application to heaps

- The last node of a heap of $n$ keys is indexed $n$ in the array.
- The first empty external node is then indexed $n + 1$.
- Don’t need to store external nodes explicitly.
Heap Example: only keys

4
5
15
16
25
14
9
12
11
7
8
6
20
Heap Example: only keys

[4, 5, 6, 15, 9, 7, 20, 16, 25, 14, 12, 11, 8]
Insertion

Goals

- Maintain three properties of heap.
- Cost \( \sim \) the height of the heap. i.e., \( O(h) = O(\log(n)) \).
Insertion

Goals

▶ Maintain three properties of heap.
▶ Cost \( \sim \) the height of the heap. i.e., \( O(h) = O(\log(n)) \).
▶ External Nodes as ”place-holders”.
▶ Heap-Order Property: for every node \( v \) other than the root, its key \( \geq \) the key of its parent.
▶ Complete Binary Trees: binary tree with height \( h \) and maximum number of nodes in all levels \( 0, \cdots, h - 1 \). In level \( h - 1 \), the internal nodes are to the left of the external nodes.
▶ Last Node: as the rightmost internal node on level \( h - 1 \).