Introduction to Algorithms and Data Structures

Heap Operations and Priority Queues
Last lecture

- Max Heap data structure

- Operations:
  - Max-Heapify
  - Build-Max-Heap

- Using these, we presented Heapsort, a sorting algorithm with worst-case running time $O(n \lg n)$. 
This lecture

• More Max Heap operations:
  
  • $\text{Max-Heap-Extract-Max}(A)$
  
  • $\text{Max-Heap-Insert}(A, v)$

• Using Heaps to implement Priority Queues
Heap operations

• **Max-Heap-Extract-Max(\(A\)):**
  \textit{Extract and return the maximum element of the heap, and also remove it from the heap.}

• **Max-Heap-Insert(\(A, v\)):**
  \textit{Insert a new element \(v\) to the heap.}
Max-Heap-Extract-Max(\(A\)):

- Max-Heap-Extract-Max(\(A\)):
  
  Extract and return the maximum element of the heap, and also remove it from the heap.
Max-Heap-Extract-Max\((A)\): 

- **Max-Heap-Extract-Max\((A)\):**
  
  Extract and return the maximum element of the heap, and also remove it from the heap.

- How can we do this?
Heapsort

Heapsort($A, n$)
1. BUILD-MAX-HEAP($A, n$)
2. for $i = n$ downto 2
4. $A.heap-size = A.heap-size - 1$
5. MAX-HEAPIFY($A, 1$)

CLRS pp 170
Heapsort

Heapsort($A, n$)
1. Build-Max-Heap($A, n$)
2. for $i = n$ downto 2
4. $A.heap-size = A.heap-size - 1$
5. Max-Heapify($A, 1$)

CLRS pp 170
Heapsort

**HEAPSORT** \((A, n)\)

1. **BUILD-MAX-HEAP** \((A, n)\)
2. **for** \(i = n \text{ downto } 2\)
3. exchange \(A[1]\) with \(A[i]\)
4. \(A.\text{heap-size} = A.\text{heap-size} - 1\)
5. **MAX-HEAPIFY** \((A, 1)\)

CLRS pp 170
Heapsort

**Heapsort**

Heapsort($A, n$)
1. **Build-Max-Heap**($A, n$)
2. for $i = n$ downto 2
   4. $A.heap-size = A.heap-size - 1$
5. **Max-Heapify**($A, 1$)

CLRS pp 170
Heapsort

Heapsort($A, n$)
1. BUILD-MAX-HEAP($A, n$)
2. for $i = n$ downto 2
4. $A$.heap-size = $A$.heap-size - 1
5. MAX-HEAPIFY($A, 1$)

CLRS pp 170
Max-Heap-Extract-Max(A):

• Max-Heap-Extract-Max(A):
  Extract and return the maximum element of the heap, and also remove it from the heap.

• How can we do this?

Max-Heap-Extract-Max(A)

1  max = A[1]
3  A.heap-size = A.heap-size − 1
4  Max-Heapify(A,1)
5  return max
Max-Heap-Extract-Max\((A)\):

- **Max-Heap-Extract-Max\((A)\):**
  Extract and return the maximum element of the heap, and also remove it from the heap.

- How can we do this?

```plaintext
Max-Heap-Extract-Max \((A)\)  Running time?
1   max = A[1]   
3   A.heap-size = A.heap-size − 1  
4   Max-Heapify\((A,1)\)   
5   return max
```
Max-Heap-Extract-Max($A$):

- Max-Heap-Extract-Max($A$):
  *Extract and return the maximum element of the heap, and also remove it from the heap.*

- How can we do this?

```
Max-Heap-Extract-Max ($A$)                      Running time?
1  max = A[1]          \(\Theta(\lg n)\)
3  A.heap-size = A.heap-size − 1
4  Max-Heapify($A, 1$)
5  return max
```
Max-Heap-Insert\((A, \nu)\)

- Max-Heap-Insert\((A, \nu)\):
  
  *Insert a new element \(\nu\) to the heap.*
Max-Heap-Insert \((A, 15)\)
Max-Heap-Insert \((A, 15)\)

Where should we add 15?
Max-Heap-Insert \((A, 15)\)

Where should we add 15?
Max-Heap-Insert \((A, 15)\)

Where should we add 15?
Max-Heap-Insert \((A, 15)\)

Where should we add 15?

The value of a node is at most the value of its parent, i.e., \(A[\text{Parent}(i)] \geq A[i]\).
Max-Heap-Insert \((A, 15)\)

Where should we add 15?

The value of a node is at most the value of its parent, i.e., \(A[\text{Parent}(i)] \geq A[i]\).
Fixing the problem
Fixing the problem

• How do we fix a tree that is “almost” a heap back to being a heap?
Fixing the problem

• How do we fix a tree that is “almost” a heap back to being a heap?

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
4       largest = l
5   else largest = i
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10  MAX-HEAPIFY(A, largest)
```
Max-Heap-Insert \((A, 15)\)

Can we use Max-Heapify here?

The value of a node is at most the value of its parent, i.e., \(A[\text{Parent}(i)] \geq A[i]\).
Max-Heap-Insert \((A, 15)\)

Can we use Max-Heapify here?

The value of a node is at most the value of its parent, i.e., \(A[i] \geq A[l] \geq A[r] \geq A[\text{largest}]\).

MAX-HEAPIFY\((A, i)\)

1. \(l = \text{LEFT}(i)\)
2. \(r = \text{RIGHT}(i)\)
3. if \(l \leq A.\text{heap-size} \text{ and } A[l] > A[i]\) then \(\text{largest} = l\)
4. else \(\text{largest} = i\)
5. if \(r \leq A.\text{heap-size} \text{ and } A[r] > A[\text{largest}]\) then \(\text{largest} = r\)
6. if \(\text{largest} \neq i\) then exchange \(A[i]\) with \(A[\text{largest}]\)
7. MAX-HEAPIFY\((A, \text{largest})\)
Max-Heap-Insert \((A, 15)\)

Can we use Max-Heapify here?

Can we use Max-Heapify here?

The value of a node is at most the value of its parent, i.e., \(A[i] \leq A[parent(i)]\).

Problem!
Max-Heap-Insert \((A, 15)\)

Can we use Max-Heapify here?

The value of a node is at most the value of its parent, i.e., \(A[\text{parent}(i)] \geq A[i]\).

**Max-Heapify**

```plaintext
1  l = LEFT(i)
2  r = RIGHT(i)
3  if \(l \leq A.\text{heap-size}\) and \(A[l] > A[i]\)
4    largest = l
5  else largest = i
6  if \(r \leq A.\text{heap-size}\) and \(A[r] > A[\text{largest}]\)
7    largest = r
8  if largest \(\neq i\)
9    exchange \(A[i]\) with \(A[\text{largest}]\)
10   MAX-HEAPIFY\((A, \text{largest})\)
```
Max-Heapify-Up \((A, 11)\):

1. If \(i > 1\) then
2. \(j = \text{Parent}(i)\)
3. If \(A[i] > A[j]\) then
4. Exchange \(A[i]\) with \(A[j]\)
5. Max-Heapify-Up \((A, j)\)
Max-Heapify-Up(A, 11):

Max-Heapify-Up (A, i)
1   if i > 1 then
2   j = Parent(i)
3   if A[i] > A[j] then
4       exchange A[i] with A[j]
5   Max-Heapify-Up (A, j)
Max-Heapify-Up \((A,11)\):

1. \(i > 1\) then \(j = \text{Parent}(i)\)
2. if \(A[i] > A[j]\) then
3. exchange \(A[i]\) with \(A[j]\)
4. Max-Heapify-Up \((A,j)\)
Max-Heapify-Up\((A,11)\):
Max-Heapify-Up(A,11):

1       if
2       then
3      if
4          exchange
5          Max-Heapify-Up

Max-Heapify-Up(A,i)
1    if \( i > 1 \) then \text{true}
2    \( j = \text{Parent}(i) \) →
3    if \( A[i] > A[j] \) then \text{true}
4    exchange \( A[i] \) with \( A[j] \)
5    Max-Heapify-Up\ (A,j)
Max-Heapify-Up\((A,11)\):

1. if \(i > 1\) then true
2. \(j = \text{Parent}(i)\)
3. if \(A[i] > A[j]\) then true
4. exchange \(A[i]\) with \(A[j]\)
5. Max-Heapify-Up \((A,j)\)
Max-Heapify-Up\((A, 11)\):

1. if \(i > 1\) then \(true\)
2. \(j = \text{Parent}(i)\)
3. if \(A[i] > A[j]\) then \(true\)
4. exchange \(A[i]\) with \(A[j]\)
5. Max-Heapify-Up \((A, j)\)
Max-Heapify-Up($A, 11$):

1. if $i > 1$ then true
2. $j = \text{Parent}(i)$ →
5. Max-Heapify-Up($A, j$)
Max-Heapify-Up\((A, 11)\):

1. if \(i > 1\) then true
2. \(j = \text{Parent}(i)\)
3. if \(A[i] > A[j]\) then true
4. exchange \(A[i]\) with \(A[j]\)
5. Max-Heapify-Up \((A, j)\)
Max-Heapify-Up\((A, 11):\)

1. if \(i > 1\) then \(\text{true}\)
2. \(j = \text{Parent}(i)\)
3. if \(A[i] > A[j]\) then \(\text{true}\)
4. exchange \(A[i]\) with \(A[j]\)
5. Max-Heapify-Up \((A, j)\)
Max-Heapify-Up\((A, 11)\):

1. \(\text{if } i > 1 \text{ then } \text{true}\)
2. \(j = \text{Parent}(i)\)
3. \(\text{if } A[i] > A[j] \text{ then } \text{true}\)
4. \(\text{exchange } A[i] \text{ with } A[j]\)
5. \(\text{Max-Heapify-Up } (A, j)\)
Max-Heap-Insert($A, v$)

- Max-Heap-Insert($A, v$):
  
  *Insert a new element $v$ to the heap.*

```
Max-Heap-Insert($A, v$)
1   $A$.heap-size = $A$.heap-size + 1
2   $A$[$A$.heap-size] = $v$
3   $i = A$.heap-size
4   Max-Heapify-Up ($A, i$)
```
Max-Heap-Insert \((A, 15)\)

Max-Heap-Insert\((A, v)\)
1. \(A.\text{heap-size} = A.\text{heap-size} + 1\)
2. \(A[A.\text{heap-size}] = v\)
3. \(i = A.\text{heap-size}\)
4. Max-Heapify-Up \((A, i)\)
Max-Heap-Insert \((A, 15)\)

Max-Heap-Insert\((A, v)\)

1. \(A.\)heap-size = \(A.\)heap-size + 1
2. \(A[A.\)heap-size\] = \(v\)
3. \(i = A.\)heap-size
4. Max-Heapify-Up \((A, i)\)
Max-Heap-Insert \((A, 15)\)
Max-Heap-Insert \((A, 15)\)

\[
\begin{align*}
1 & A.\text{heap-size} = A.\text{heap-size} + 1 \\
2 & A[A.\text{heap-size}] = v \\
3 & i = A.\text{heap-size} \\
4 & \text{Max-Heapify-Up} (A, i)
\end{align*}
\]
Max-Heap-Insert \((A, 15)\)

Max-Heap-Insert\((A, v)\)

1. \(A\).heap-size = \(A\).heap-size + 1
3. \(i = A\).heap-size
4. Max-Heapify-Up \((A, i)\)
Max-Heap-Insert \((A, 15)\)

Max-Heap-Insert\((A, v)\)
1. 11 \(A\).heap-size = \(A\).heap-size + 1
3. 11 \(i = A\).heap-size
4. Max-Heapify-Up \((A, i)\)
Max-Heap-Insert \((A, 15)\)

Max-Heap-Insert\((A, v)\)
1. \(A.\text{heap-size} = A.\text{heap-size} + 1\)
3. \(i = A.\text{heap-size}\)
4. Max-Heapify-Up \((A, i)\)
Max-Heap-Insert$(A, v)$

- Max-Heap-Insert$(A, v)$:  
  *Insert a new element $v$ to the heap.*

```
Max-Heap-Insert$(A, v)$
1  $A$.heap-size = $A$.heap-size + 1
2  $A[A$.heap-size$] = v$
3  $i = A$.heap-size$
4  $\text{Max-Heapify-Up}(A, i)$
```
Max-Heap-Insert \((A, \, v)\)

- **Max-Heap-Insert\((A, \, v)\):**
  
  *Insert a new element \(v\) to the heap.*

  
  \[
  \text{Max-Heap-Insert}(A, \, v) \\
  1 \quad A.\text{heap-size} = A.\text{heap-size} + 1 \\
  2 \quad A[A.\text{heap-size}] = v \\
  3 \quad i = A.\text{heap-size} \\
  4 \quad \text{Max-Heapify-Up}\ (A, \, i)
  \]

What is the running time of Max-Heap-Insert?
Max-Heap-Insert($A, v$)

- Max-Heap-Insert($A, v$):
  
  Insert a new element $v$ to the heap.

Max-Heap-Insert($A, v$)

1. $A$.heap-size = $A$.heap-size + 1
3. $i = A$.heap-size
4. Max-Heapify-Up ($A, i$)

What is the running time of Max-Heap-Insert?

What is the running time of Max-Heapify-Up?
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node $i$.

$$T(h) \leq (h + 1) \cdot O(1)$$

$$= O(h) = O(\lg n)$$
Max-Heapify running time

What is the cost of an execution of Max-Heapify?

All “standard” operations can be done in $O(1)$ time.

Plus the time needed for the recursive call of Max-Heapify on the child of node $i$.

\[
T(h) \leq (h + 1) \cdot O(1) = O(h) = O(\lg n)
\]

\begin{Verbatim}
MAX-HEAPIFY(A, i)  T(h)
1  l = LEFT(i)  O(1)
2  r = RIGHT(i)  O(1)
4    largest = l  O(1)
5  else largest = i  O(1)
7    largest = r  O(1)
8  if largest ≠ i O(1)
10  MAX-HEAPIFY(A, largest)  T(h − 1)
\end{Verbatim}

\[
T(h) \leq \begin{cases} 
T(h − 1) + O(1), & \text{if } h \geq 1 \\
O(1) & \text{if } h = 0
\end{cases}
\]

Argument for Max-Heapify-Up almost identical!
Max-Heap-Insert\((A, v)\) (without recursion)

Max-Heap-Insert\((A, v)\):

*Insert a new element v to the heap.*

```
Max-Heap-Insert\((A, v)\)
1     A.heap-size = A.heap-size + 1
2     A[A.heap-size] = v
3     i = A.heap-size
4     while \((i \neq 1 \text{ and } A[i] > A[\text{Parent}(i)])\) do
5         exchange \(A[i]\) with \(A[\text{Parent}(i)]\)
6         i = Parent(i)
```
Priority Queues
Priority Queues

• Priority queue: A data structure that maintains
Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements $S$. 
Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements $S$.
  - Each with an associated value, $\text{key}(v)$. 
Priority Queues

• Priority queue: A data structure that maintains

  • A set of elements $S$.

  • Each with an associated value, $\text{key}(v)$.

  • The values denote priorities.
Priority Queues

- Priority queue: A data structure that maintains
  - A set of elements $S$.
  - Each with an associated value, $\text{key}(v)$.
  - The values denote priorities.
  - For Max-Priority Queues, the elements with the largest values are those with the highest priority.
Priority Queues
Priority Queues

- Example: Scheduling processes on a computer.
Priority Queues

- **Example:** Scheduling processes on a computer.
  - Each process has a priority or urgency.
Priority Queues

• **Example:** Scheduling processes on a computer.
  - Each process has a priority or urgency.
  - Processes don’t arrive in order of priorities.
Priority Queues

- **Example:** Scheduling processes on a computer.
  - Each process has a priority or urgency.
  - Processes don’t arrive in order of priorities.
  - From the set of active processes, we need to find that with the highest priority and run it.
Priority Queue Operations
Priority Queue Operations

- $\text{Insert}(Q, \nu)$ inserts a new item $\nu$ in the priority queue.
Priority Queue Operations

- **Insert**\( (Q, v) \) inserts a new item \( v \) in the priority queue.

- **FindMax**\( (Q) \) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).
Priority Queue Operations

- \textbf{Insert}(Q, v)\ inserts a new item \( v \) in the priority queue.

- \textbf{FindMax}(Q)\ finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).

- \textbf{ExtractMax}(Q)\ finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.
Implementing Priority Queues
Implementing Priority Queues

• Approach 1: Store the elements in an array/list. Also maintain a pointer for the max element.
Implementing Priority Queues

• **Approach 1:** Store the elements in an array/list. Also maintain a pointer for the max element.

  • How long does it take to find the max element?
Implementing Priority Queues

• **Approach 1:** Store the elements in an array/list. Also maintain a pointer for the max element.
  
  • How long does it take to find the max element?

  • $O(1)$
Implementing Priority Queues

• **Approach 1:** Store the elements in an array/list. Also maintain a pointer for the max element.

  • How long does it take to find the max element?
    - \( O(1) \)

  • How long does it take to insert a new element?
Implementing Priority Queues

- **Approach 1**: Store the elements in an array/list. Also maintain a pointer for the max element.

  - How long does it take to find the max element?

    - $O(1)$

  - How long does it take to insert a new element?

    - Need to update the max pointer, hence $O(n)$. 
Implementing Priority Queues
Implementing Priority Queues

- **Approach 2:** Store the elements in a *sorted* array/list.
Implementing Priority Queues

• **Approach 2:** Store the elements in a *sorted* array/list.

• How long does it take to find the max element?
Implementing Priority Queues

- Approach 2: Store the elements in a *sorted* array/list.
- How long does it take to find the max element?
  - $O(1)$
Implementing Priority Queues

• **Approach 2:** Store the elements in a *sorted* array/list.

• How long does it take to find the max element?
  • $O(1)$

• How long does it take to insert a new element?
Implementing Priority Queues

- **Approach 2**: Store the elements in a *sorted* array/list.

- How long does it take to find the max element?
  - $O(1)$

- How long does it take to insert a new element?
  - We need to find the right position to insert it in the array - $O(1 \log n)$ using binary search.
Implementing Priority Queues

• Approach 2: Store the elements in a *sorted* array/list.

• How long does it take to find the max element?
  • $O(1)$

• How long does it take to insert a new element?
  • We need to find the right position to insert it in the array - $O(1g\ n)$ using binary search.
  • We still need to insert it, which means moving all the later elements one position to the right - $O(n)$. 
Implementing Priority Queues
Implementing Priority Queues

- **Approach 3:** Use a Max Heap.
Implementing Priority Queues

• **Approach 3:** Use a Max Heap.

• How long does it take to find the max element?
Implementing Priority Queues

• **Approach 3:** Use a Max Heap.

• How long does it take to find the max element?
  
  • $O(1)$
Implementing Priority Queues

• **Approach 3:** Use a Max Heap.

• How long does it take to find the max element?
  
  • $O(1)$

• How long does it take to insert a new element?
 Implementing Priority Queues

• **Approach 3:** Use a Max Heap.

• How long does it take to find the max element?
  
  • $O(1)$

• How long does it take to insert a new element?
  
  • $O(\lg n)$ via Max-Heap-Insert.
Priority Queue Operations

- **Insert**\((Q, v)\) inserts a new item \(v\) in the priority queue.

- **FindMax**\((Q)\) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).

- **ExtractMax**\((Q)\) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.
Priority Queue Operations

• **Insert**\((Q, v)\) inserts a new item \(v\) in the priority queue.  \(O(\lg n)\)

• **FindMax**\((Q)\) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it).

• **ExtractMax**\((Q)\) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.
Priority Queue Operations

- **Insert**\((Q, v)\) inserts a new item \(v\) in the priority queue. \(O(\lg n)\)

- **FindMax**\((Q)\) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it). \(O(1)\)

- **ExtractMax**\((Q)\) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue.
Priority Queue Operations

- **Insert**\((Q, v)\) inserts a new item \(v\) in the priority queue. \(O(\lg n)\)

- **FindMax**\((Q)\) finds the element with the maximum priority (the highest value) in the priority queue and returns it (but does not remove it). \(O(1)\)

- **ExtractMax**\((Q)\) finds the element with the maximum priority (highest value) in the priority queue, returns it, and deletes it from the queue. \(O(\lg n)\)
Max-heaps notes

• Python `heapq` library.

• They use Min-heaps rather than Max-heaps
  • So do Roughgarden and KT.
  • Arrays indexed from 0 (these slides and CLRS/KT index from 1).

• Names of operations are different
  • e.g., Their `heapify` is basically our `Build-Max-Heap`, our `Max-Heapify` is part of their `Heapop` (which is the equivalent of our `Max-Heap-Extract-Max`).
Reading

• CLRS 6.5
  
  • Notes: Uses max-heaps but presents the heap operations in the context of priority queues first, using an additional increase key operation.

• KT 2.5.
  
  • Notes: Very close to the exposition of these slides. Uses a min heap rather than a max heap, and further implements a general delete operation.

• Roughgarden 10.2, 10.5
  
  • Notes: Uses a min heap rather than a max heap. The operation heapify builds a heap from scratch, so it is like Build-Min-Heap. The operation that restores an “almost” heap into a heap is part of ExtractMin.