Heaps and Heapsort

Module 3: Efficient Sorting and Selection

Overview

So far we have discussed tools necessary for analyzing algorithms (asymptotic notation, summations and recurrences) and we have seen a couple of sorting algorithms at work. Today we discuss a generic data structure called *priority queue*, and its implementation with a heap. The heap will lead to a different algorithm for sorting, called *heapsort*.

Goals:

- Understand the interface of a Priority Queue
- Understand how heaps are defined, the operations supported by a heap (Find-Min, Delete-Min, Insert, Heapify, Buildheap) and their $\Theta()$ bounds
- Understand Heapsort
- Be able to use the heap as a tool to solve new problems

1 The Priority Queue

- Today's topic is the *heap*, which is the standard implementation of a generic data structure called a *priority queue*. We start by introducing the priority queue.
- What is a priority queue? It's basically an extension of the first-in-first-out queue that you saw in Data Structures. In a regular queue all elements are equal, while in a priority queue the elements have designated *priorities* and are handled according to their priorities.
- For example: the elements could be people containing first name, last name and age, and the priority of a person could be for e.g. the age. Generally speaking, imagine an element as an object containing several fields, and one of them is designated as the priority.
- The setup is the following: we have a set of elements, each with a given priority, and we want to be able to do two things: add new elements to the set, and delete the element in the set with the "best priority" — and we want to do both operations efficiently.
- To summarize, a priority queue supports the following operations on a set S of elements:
  - Insert a new element e to S
  - Delete the **best** element in S
  - also: Peek at the **best** element in S, but don't delete it

- The **best** element is the one having the **best** priority. Depending on the specific application and what priorities represent, sometimes smaller priorities are better than larger priorities (for e.g. when priority represents the error), other times larger priorities are better than smaller priorities (for e.g. when priority represents seniority). We define **min-priority queues** and **max-priority queues**, respectively.
- A **max-priority queue** is used when larger priorities are better; it supports the following basic operations:
  - INSERT(e, S): Insert a new element e in S
  - FIND-MAX (S): Return the element with **max** priority in S
  - DELETE-MAX (S): Delete the elements with **max** priority in S
- Symmetrically, a **min-priority queue** is used when smaller priorities are better; it supports the following basic operations:
  - INSERT(e,S): Insert a new element e in S
  - FIND-MIN(S): Return the element with **min** priority in S
  - DELETE-MIN(S): Delete the elements with **min** priority in S

1.1 Why priority queues?

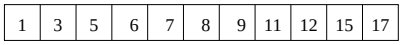
- Priority queues are fundamental structures and are frequently used as building blocks in algorithms. For example, priority queues are frequently used to solve applications that handle events: if an application requires to process a dynamic set of events (events need to be inserted), and process the events in order, you'll probably want to use a priority queue. For e.g. the OS (operating system) job scheduler keeps track of the jobs to be run in a priority queue.
- It is convenient to think of priority queues as generic structures rather than equate priority queues with a specific implementation. This is helpful because it allows to focus on the high-level ideas

1.2 Using a priority queue to sort

- Problem: Assume we are given a set S of elements that need to be sorted. Can we sort using a priority queue?
- Yes! We can sort using a priority queue as follows:
  - Set the priority of the elements in S as the key/value by which we want to sort, and initialize a PQ as empty,
  - Insert all elements in S into the PQ using INSERT
  - Delete all elements in PQ in order using DELETE-MIN. Voila!
- Analysis: How long does this take, assuming the size of S is n elements?
  Answer: $O(n\cdot$ INSERT $+ n\cdot$ DELETE-MIN $)$

1.3 A first shot at implementing a priority queue

- Now that we know what a priority queue is supposed to do, the question is: How to implement a priority queue? We'll focus on min-priority queues. Max-priority queues are symmetrical.
- For simplicity let's assume the set S consists of numbers, and the priority is the value.
- As usual, we denote the number of elements in S by n
- **PQ as ordered array:** The first implementation that comes to mind is an ordered array: we keep the elements in S in an array ordered by priority.



  - FIND-MIN can be performed in $O(1)$ time
  - DELETE-MIN takes $\Theta(n)$ time because it needs to shift all elements over to the left
  - INSERT takes $\Theta(n)$ time in the worst case since it needs to find the right place to insert and then shift the elements to the right to make space for the new element

  Note that using this PQ implementation to sort leads to $O(n\cdot$ INSERT $+ n\cdot$ DELETE-MIN $) = \Theta(n^2)$

- **PQ as unordered array:** If we don't enforce that the array is ordered, then inserts are faster, because the new element can be added at the end (however in the worst case it's still $\Theta(n)$ because it needs to re-allocate the array and copy all elements over). Find-Min and Delete-Min would take $\Theta(n)$ time because they need to traverse the unordered array and search for the minimum.

  Note that using this PQ implementation to sort leads to $O(n\cdot$ INSERT $+ n\cdot$ DELETE-MIN $) = \Theta(n^2)$

- **PQ as an ordered linked list:** To avoid the limitations of arrays (namely the $O(n)$ expansion/compression cost) we could use an ordered doubly-linked list.

  - FIND-MIN and DELETE-MIN are now constant time;
  - but INSERT can still take $\Theta(n)$ time worst-case because it first needs to locate where in the list to insert the new element.

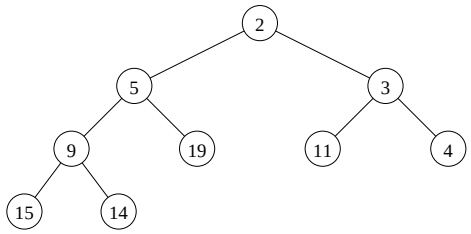
  Note that using this PQ implementation to sort leads to $O(n\cdot$ INSERT $+ n\cdot$ DELETE-MIN $) = \Theta(n^2)$

Naturally, we ask ourselves: Can we do better?

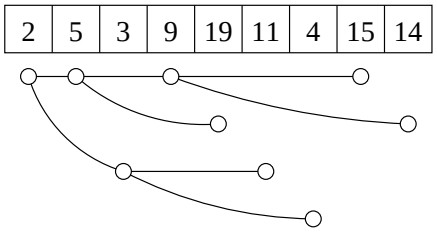
Yes! Now we'll see a simple and elegant structure called the **heap** which can be used as a priority queue, with $O(\lg n)$ time per operation.

2 The Heap

- There are *min-heaps*, and *max-heaps*, defined symmetrically. We'll focus on min-heaps.
- A Min-Heap is:
  - An array which we view as representing a perfectly balanced binary tree, where the lowest level can be incomplete, and it's filled from left-to-right;
  - (Heap property:) Such that for all nodes v, priority(v)≥priority(parent(v)).
- Here's a Min-heap example. Note that it's a perfectly balanced tree (except for the last level), and that every node is smaller than it's children.



- In a heap we know that each node is smaller than its children, which are smaller than their children, and so on. Where does the smallest element in the heap live? That's right, it must be in the root!
- The beauty of heaps is that although they are trees, they can be stored as arrays. The elements in the heap are stored level-by-level, left-to-right in the array. There is no need for nodes to store pointers to children (as in a tree); for every element i in the array, the indices of its parent and its children can be computed in constant time.
- Example: The heap above is stored as an array as follows. We assume the heap starts at index 1 (the element at index 0 is not used).

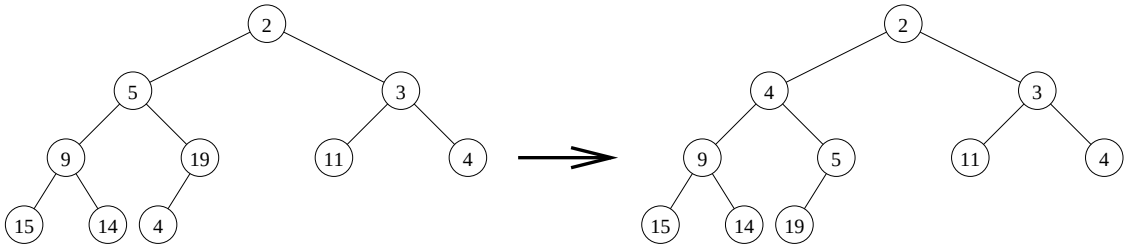


  - the left a child of node at index i is the node at index $2i$: left(i) = $2i$
  - the right child of node at index$i$ is the node at index $2i + 1$: right(i) = $2i + 1$
  - the parent of node at index i is the node at index $\lfloor \frac{i}{2} \rfloor$: parent($i$) = $\lfloor \frac{i}{2} \rfloor$
- It can be shown that the height of a heap of n elements is $\Theta(\log n)$

2.1 Inserting in a heap

- To insert a new element e in the heap, we grow the array by one and add e to the end of the array; this corresponds to adding a new leaf in leftmost possible position on lowest level. Then repeatedly swap element with element in parent node until heap order is reestablished (this is sometimes refered to as UP-HEAPIFY).

- Example: Insert(4): In the image on the left the new element 4 is placed at the end of the array, on the last level. Now 4 is smaller than its parent 19, so they are swapped. Then 4 is compared to its new parent 5, and they are swapped. Now 4's new parent is 2, which satisfies heap property so insertion is all done.



- Why is this correct? We need to argue that inserting a new element as described above produces a heap. Let's refer to the example above for simplicity. We shart with a heap; After we add 4 on the last level, the only place where heap property may be violated is between 4 and its parent, 19. We swap them. Now the only place where the heap property may be violated is between 4 and its new parent, 5. Since 4 is smaller then 5, we swap them. This maintains heap property to the left of 5, because we replace 5 with something that's smaller. And it maintains heap property to the right of 5, because we know that 5 was smaller then its right child, so it's ok to make it the root of its right child.

- Analysis: Inserting a new element in a heap triggers swapping elements on the way from the leaf up to the root. In the worst case it can propagate all the way to the root (when inserting an element smaller than the root). Since the height of a heap is $\Theta(\lg n)$, an Insert takes $\Theta(\lg n)$ in the worst case.

2.2 PEEK-MIN in a heap

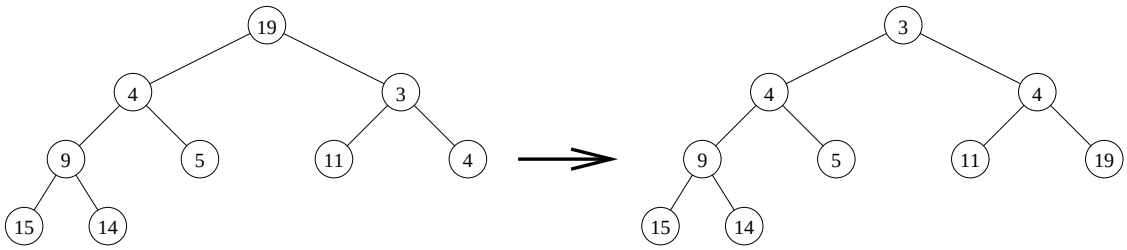
- Just return the element at index 1 in the array; takes $O(1)$ time.

2.3 DELETE-MIN in a heap

- The min element is the root. To delete it, we delete the element in the root (ie at index 1 in the array), then shrink the array and move the last element (ie rightmost leaf on lowest level) to the root. This will violate heap property at the root. To restore heap order, swap element with the smaller of its two children, and repeat from that child; this operation is called Heapify-down().

- Example: Consider the heap above. To delete the element at the root, 2, we place the last element, 19, at the root and then "heapify" down: we swap 19 with the smaller of its two

children, in this case 3. Next we swap 19 with the smaller of 11 and 4, which is 4. The resulting heap is shown to the right.



- Why is this correct? We start with the last element being moved to the root. Its left and right subtrees are both heaps. Then we swap the root with the smaller of its two children. Let's assume this is the left child. The swap restores heap property at the root, and the right child is unchanged, so it's still a heap. But the swap may have violated heap property in the left child — now we are in the same situation as when we started but one node lower in the heap.

- Analysis: This involves swaps from the root on a path down to a leaf. In the worst case the swaps will continue all the way down to the last level of the heap. Since the height of the heap is $\Theta(\lg n)$, the worst-case of Delete-Min is $\Theta(\lg n)$

2.4 Other operations: CHANGE and DELETE in a heap

Changing the priority of a given node or deleting a given node can be handled similarly in $O(\log n)$ time. However, it's important to realize that we can delete or update nodes in a heap if we are given their index in the array. For e.g. we cannot say "delete the node with priority 37" because we cannot search efficiently in a heap! But we can say "delete the node at index 5".

2.5 Converting an array to a heap

- The last operation we discuss is the following: assume we have an array A of n elements and we'd like to create a heap with them.

- Of course, we can call INSERT() repeatedly, for a total of $O(n \lg n)$.

- Can we do better?

- Yes, we can do better, and it turns out we can turn the array A into a heap without using additional space, and in linear time. It' s a neat and simple algorithm. To write it, let's recall the procedure called Heapify-down(i) which we used in the Delete-Min operation. Heapify-down(i) is called on a node i in the heap. The assumption is that the heap property may be violated at node i, but the children of i are both (valid) heaps. Heapify-down(i) swaps node i with the smaller of its two children, and recurses on that child. When Heapify-down (i) is complete, the tree rooted at node i is a heap.

  To make an array into a heap, the idea is to call Heapify-down() on all nodes, going bottom-up in the tree.

BUILDHEAP-smart (A)

- For i = n/2 down to 1: HEAPIFY-DOWN(i)

- Correctness: Why does this work? Note that we call Heapify-down going backwards in the array from $i = n/2$ down to 1, which corresponds to going bottom up in the heap; this means that when calling it on a node on level k, all trees rooted at level $k-1$ have been already made into heaps, so the assumptions of Heapify-down are fullfilled.

(optional) Analysis: In summary the procedure to build a heap as described above takes $O(n)$ time. This is totally optional, but here are the key points if you are curious.

- The "height" of a node in the heap is the number of levels below it (the leaves are at height 0, the level above is at height 1, and so on, the root is at height $\log n$)
- The cost of HEAPIFY(i) on a node i at height h is $O(h)$
- In a heap of n elements there are $\frac{n}{2^h}$ elements at height h
- Thus the total cost is $\sum_{i=1}^{\log n} h \cdot \frac{n}{2^h}$
- it can be shown that $\sum_{i=1}^{\log n} \frac{h}{2^h} = O(1) \Longrightarrow$ the total buildheap cost is $\Theta(n)$

3 Heapsort

- Finally, we are ready to sort with a heap!
- Assume we start with an array A of n elements. We can insert the elements into a heap, one at a time by calling Insert, and then call Delete-Min on the heap repeatedly. Since both operations run in $O(\lg n)$ time, this would take $O(n \lg n)$ time. Hooray! It's another sort that runs in $O(n \lg n)$!
- This is great, but there is one place where this can be improved, namely, it's not "in-place". It uses the heap in addition to the input array A. Could we sort with a heap in place?
- Yes, an in-place sorting algorithm with a heap is possible, and it's this algorithm that's referred to as HEAPSORT
- First, we can make the array A into a heap by using the smart Buildheap algorithm discussed above. This is in place and runs in $O(n)$ time.
- Now the array A is a heap, and we can delete the smallest element repeatedly. As we delete elements from the heap, the array shrinks, and .... we don't want to use any extra space, so we can put the elements that we deleted at the end of the array. But.... that means we get the array sorted in reverse order... almost there...
- Wait! To get the array sorted in increasing order... it means we have to repeatedly delete not the min element, but the max element. But, we can do that with a Max-heap! And so we got Heapsort:

-
```
Heapsort(A)
    Convert A into a max-heap
    //Repeatedly Delete-Max and put it at the end of the array
    for i=0 to n-1: A[n-i] = DELETE-MAX(A)
```

- Analysis: creating the heap takes $O(n)$ time, and deleting the max n times takes $\Theta(n \lg n)$.
- Time to go through an example!

4 Final notes on heaps

- Take a minute to think about the difference between heaps and binary search trees (BSTs)
- Does a BST support Insert and Delete-Min?

  Answer: Yes, and if the BST is balanced, these operations also take $O(\lg n)$ time
- A heap supports only a subset of the operations supported by a BST in $O(\lg n)$ time. For e.g. a BST also supports FIND-MAX, DELETE-MAX, Search, find-successor, find-predecessor, etc.
- The *heap property* and the *binary search tree property*: are similar yet different. In a heap we know that each node is smaller than its children, which are smaller than their children, and so on. The binary search tree property says that a node is larger than its left subtree, and smaller than its right subtree, therefore it enforces a left-to-right-order.
- The heap is basically an array, not a tree, so no child pointers; uses less space
- A heap corresponds to a balanced tree, so no need for balancing operations; the constants in $O(\lg n)$ are smaller
- But, a heap cannot support SEARCH efficiently: we know the root is the smallest element, but we do not know whether to go to the left child or right child when searching for a key. A search in a heap is basically a linear search and runs in $O(n)$. Also, heaps cannot support other ops efficiently (find-successor, find-predecessor).
- The heap we discussed is a binary heap (a node has two children). They can be generalized to ternary heaps (each node has 3 children instead of 2), and so on.