CS 225 Spring 2019 :: TA Lecture Notes 4/1 Heaps III + Disjoint Sets

By Wenjie

BuildHeap Runtime

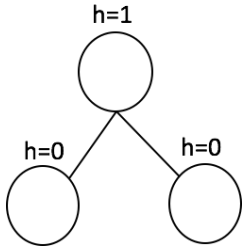
- There are three methods:
  - Sort the array: O(n log n)
  - Repeated insert (heapify-up): O(n log n)

```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = 2; i <= size_; i++) {
4      heapifyUp(i);
5    }
6  }
```

  - Heapify-down from half of the array: O(n)

```
1  template <class T>
2  void Heap<T>::buildHeap() {
3    for (unsigned i = parent(size); i > 0; i--) {
4      heapifyDown(i);
5    }
6  }
```

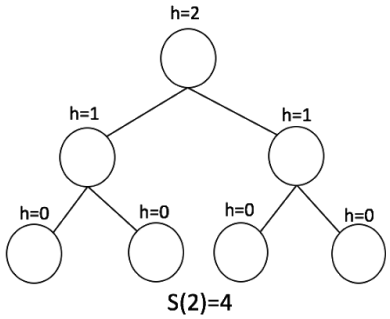
- **Theorem: BuildHeap based on heapify-down takes O(n) time.**
  - Idea:
    - Notice that, **heapify-down(i)** on the node **i** takes time: *the height of* **i**
    - Then the sum of all heights in the tree will be the running time
    - Create a formula to calculate that for all heaps of height **h**
    -
  - Define: S(h) = sum of the heights of all nodes in a perfect tree of height **h**
    - S(0) = 0; single node has height 0.
    - S(1) = 1



    - S(2) = 4

CS 225 Spring 2019 :: TA Lecture Notes
4/1 Heaps III + Disjoint Sets

By Wenjie



- Recurrence: $S(h) = 2 * S(h-1) + h.$
  - the root has height **h**
  - the children has total height **S(h-1)**
- Solve the recurrence, we have
  - **S(h) =** $2^{h+1} - h - 2$
  - This can be done using characteristic equations, or generating functions, or WolframAlpha :)

- Prove the recurrence by induction
  - Base Case:
    - $S(0) = 2^{0+1} - 0 - 2 = 0.$
    - $S(1) = 2^{1+1} - 1 - 2 = 1.$
  - Inductive Hypothesis:
    - $S(k) = 2^{k+1} - k - 2,\ k < h.$
  - Inductive Step:
    - $$\begin{aligned} S(h) &= 2 * S(h-1) + h \\ &= 2 * [2^{(h-1)+1} - (h-1) - 2] + h \\ &= 2^{h+1} - 2 * h + 2 - 4 + h \\ &= 2^{h+1} - h - 2. \ \blacksquare \end{aligned}$$
- Actual Runtime analysis: we know $S(h)$ is the runtime
  - First, $S(h) = 2^{h+1} - h - 2 = O(2^h).$
  - Second, $h = log(n).$
  - Therefore, $O(2^h) = O(2^{lgn}) = O(n).$

- Now, we can find the minimum element in $O(n)$, and create a heap out of it! Better than a Balanced BST

CS 225 Spring 2019 :: TA Lecture Notes
4/1 Heaps III + Disjoint Sets

By Wenjie

Heap Sort

- First, build a heap using the array
  - $O(n)$.
- Call remove minimum until the heap is empty
  - $O(n\ log(n))$ time.
  - When we call remove minimum, we will not actually remove the element, we will just place it at the end of the array. This will allow us to sort in place.
- Finally, we may want to reverse the list
  - $O(n)$.
- This algorithm uses 0 extra memory: it is an in-place $O(n\ log(n))$ (the best we can do) sort.

Disjoint Sets

- Partition all data into equivalence classes
  - every element exists in exactly one set
  - every set is an equitant representation of itself
    - Mathematically: $4 \in [0]_R \rightarrow 8 \in [0]_R$
    - Programmatically: find(4) == find(8)
- The operation **(find)** is our equivalence relation

Disjoint Sets ADT

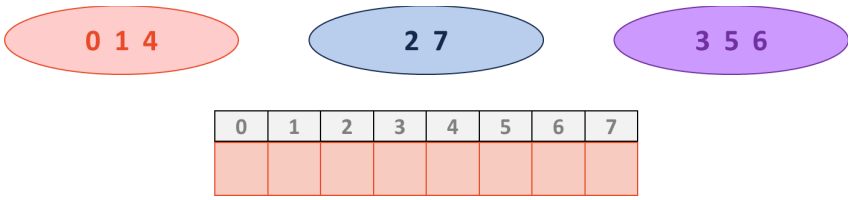
- To maintain a collection $S = \{s_0, s_1, \ldots s_k\}$ - *a set of disjoint sets*, where each set has a representative member.
- API:
  - `void makeSet(const T & t);`
    - create a set with one element
  - `void union(const T & k1, const T & k2);`
    - set + set
  - `T & find(const T & k);`
    - find the representative element

CS 225 Spring 2019 :: TA Lecture Notes
4/1 Heaps III + Disjoint Sets

By Wenjie

Implementations

- Idea:
  - Map elements to an array.
  - Indices of the array correspond to the elements in the sets.
  - Values in the array correspond to the representative members of the sets.



- **Implementation 1**
  - Choose a unique representative element for each set. It makes sense to choose one of the elements in the set.
    - $[0]_R \rightarrow \{0, 1, 4\}$
    - $[7]_R \rightarrow \{2, 7\}$
    - $[3]_R \rightarrow \{3, 5, 6\}$
  - Match identity of the set to the index of each element:

0	1	2	3	4	5	6	7
0	0	7	3	0	3	3	7

- `Find(k)`
  - Looking up the value at index **k**
  - Takes O(1)
- `Union(k1, k2)`
  - The representative element for all the elements in **k1** and **k2** must be the same.
  - Eg: Union(0, 7), we will get one set → $\{0, 1, 4, 2, 7\}$
  - Update the representatives: O(n)
    - We have to go through the entire array - bad!

0	1	2	3	4	5	6	7
0	0	0	3	0	3	3	0