Data Structures and Algorithms

Mohammad GANJTABESH

Department of Computer Science,
School of Mathematics, Statistics and Computer Science,
University of Tehran,
Tehran, Iran.

mgtabesh@ut.ac.ir

Dept. of Computer Science
University of Tehran

Heap → min → – Access to the min value is $O(1)$
Heap → max → – Adjusting the heap is $O(\log_2 n)$

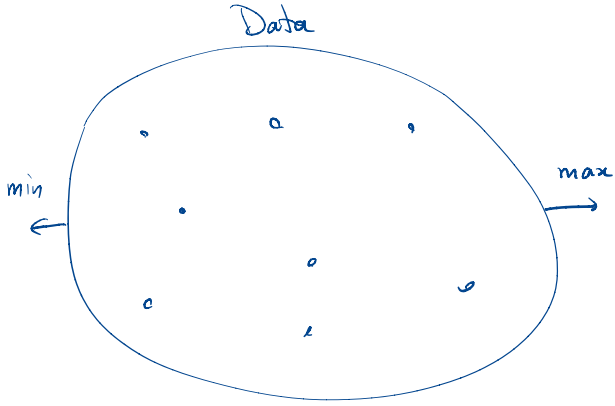
Double Ended Priority Queue (Min-Max-Heap)

Data Structures and Algorithms
Undergraduate course

Mohammad GANJTABESH
mgtabesh@ut.ac.ir

Dept. of Computer Science
University of Tehran

Motivations



- Access to the min and max in constant time.
- Adjusting the structure would be of $O(\log_2 n)$.

Operations and Naive Implementations

↳ Min, Max, Add, Remove Min, Remove Max, and other adjusting methods

Structure	Min/Max	Add	Remove Min / Remove Max	Adjusting
Array	$O(n)$	$O(1)$	$O(n)$	✗
Sorted Array	$O(1)$	$O(n)$	$O(n)$ / $O(1)$	✗
Linked List	$O(n)$	$O(1)$	$O(n)$	✗
Sorted Linked List	$O(1)$	$O(n)$	$O(1)$ / $O(n)$	✗

Operations and Naive Implementations

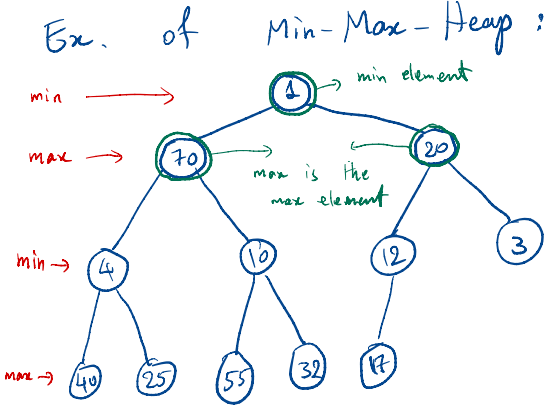
- Duplication in data.
- Add/Remove is not efficient.

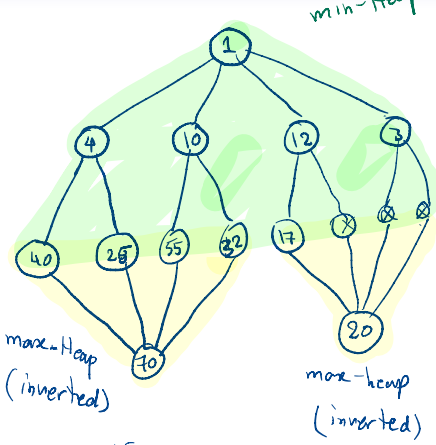
Tree Implementation: Properties

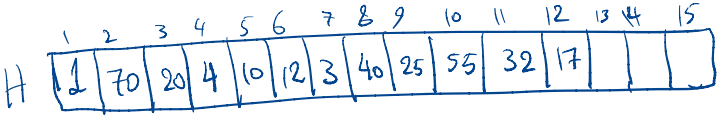
Min-Max-Heap

- has a heap structure (complete binary tree ⇓ storing in array is appropriate)
- the levels are alternatively min and max
- Access to the min and max could be done in constant time.
- Data in min (max) level is smaller (larger) than its children.

Tree Implementation: Properties







capacity = 15
size = 12

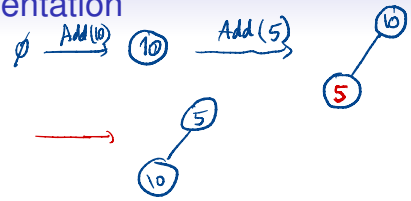
Tree Implementation

```
class Min-Max-Heap {
    int *H;
    int capacity;
    int size;
    Min-Max-Heap (int n){
        H = new int[n];
        size = 0;
        capacity = n;
    }

    // methods.

}

// Array is started by index 1
```

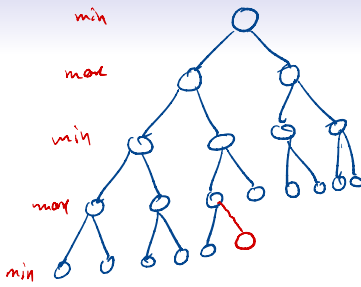
```
Min () {
    if (size = 0)  // Heap is empty.
    return (H[1]);
}
Max () {
  if (size = 0)  // Heap is empty.
  if (size = 1)  return (H[1]);
  if (size = 2)  return (H[2]);
  return (max(H[2], H[3]));
}
Level (i) {
    if( ⌊log(i)⌋ % 2 = 0)
                return (min)
    return (max)
}
```

Tree Implementation

```
Add(x) {
  if (size = capacity) // Heap is full.
  H[size] ← x;
  size++;
  if(size = 1) return;
  p ← size/2;
  if ( level(size) = min) {
      if( H[size] > H[p]) {
          swap (H[size], H[p]);
          FixUp(p);
      }
      else {
          FixUp(size)
      }
  }
  else {  // level(size) = max
      if( H[size] < H[p]) {
          swap(H[size], H[p]);
          FixUp(p);
      }
      else {
          FixUp(size);
      }
  }
```



Tree Implementation




```
FixUp(i){
    if(Level(i) = min)
        FixUpMin(i);
    else
        FixUpMax(i);
}
```

```
FixUpMin(i){
    gp ← i/4 ; // index of grandparent.
    if(gp > 0){
        if(H[i] < H[gp]){
            swap(H[i], H[gp]);
            FixUpMin(gp);
        }
    }
}
```

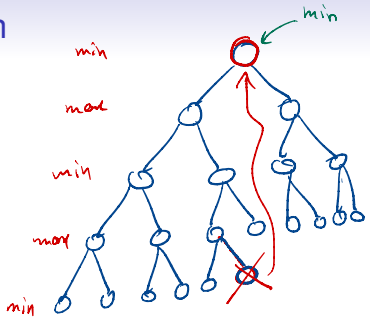
```
FixUpMax(i){
    gp ← i/4;
    if(gp > 0){
        if(H[i] > H[gp]){
            swap(H[i], H[gp]);
            FixUpMax(gp);
    } }
}
```

Tree Implementation

```
RemoveMin(){
  if(size=0) // Heap is empty.
  min ← H[1];
  H[1] ← H[size];
  size--;
  FixDown(1);
  return(min);
}
FixDown(i){
  if(Level(i) = min)
      FixDownMin(i);
  else
      FixDownMax(i);
}
```

FixDownMax() is similar to

```
FixDownMin(i){
  if(i has children){
    m ← index of smallest child or grandchild of i
    if(m is grandchild){
      if(H[m] < H[i]){ swap(H[m], H[i]);
        if(H[m] > H[Parent(m)])
          swap(H[m], H[Parent(m)]);
      } FixDownMin(m);
    }
    else if(H[m] < H[i]){ // m is child
      swap(H[m], H[i]);
    }
  }
}
```



Tree Implementation

```
RemoveMax(){
    if(size = 0) // Heap is empty.
    if(size = 1){   // root is the max.
        max ← H[1];
        size -- ;
        return (max);
    }
    if(size = 2){
        max ← H[2] ;
        size -- ;
        return (max) ;
    }
    if(H[2] > H[3])  // H[2] is max
        max ← H[2];
        H[2] ← H[size];
        size -- ;
        FixDown(2);
        return (max);
    }
    else{      // H[3] is max
        max ← H[3];
        H[3] ← H[size];
        size -- ;
        FixDown(3);
        return (max);
    }
}
```

Tree Implementation

Example :