

Heap Algorithms

PARENT(A, i)

// Input: A : an array representing a heap, i : an array index
// Output: The index in A of the parent of i
// Running Time: $O(1)$
1 **if** $i == 1$ **return** NULL
2 **return** $\lfloor i/2 \rfloor$

LEFT(A, i)

// Input: A : an array representing a heap, i : an array index
// Output: The index in A of the left child of i
// Running Time: $O(1)$
1 **if** $2 * i \leq \text{heap-size}[A]$
2 **return** $2 * i$
3 **else return** NULL

RIGHT(A, i)

// Input: A : an array representing a heap, i : an array index
// Output: The index in A of the right child of i
// Running Time: $O(1)$
1 **if** $2 * i + 1 \leq \text{heap-size}[A]$
2 **return** $2 * i + 1$
3 **else return** NULL

MAX-HEAPIFY(A, i)

// Input: A : an array where the left and right children of i root heaps (but i may not), i : an array index
// Output: A modified so that i roots a heap
// Running Time: $O(\log n)$ where $n = \text{heap-size}[A] - i$
1 $l \leftarrow \text{LEFT}(i)$
2 $r \leftarrow \text{RIGHT}(i)$
3 **if** $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
4 $\text{largest} \leftarrow l$
5 **else** $\text{largest} \leftarrow i$
6 **if** $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$
7 $\text{largest} \leftarrow r$
8 **if** $\text{largest} \neq i$
9 exchange $A[i]$ and $A[\text{largest}]$
10 MAX-HEAPIFY($A, \text{largest}$)

BUILD-MAX-HEAP(A)

// Input: A : an (unsorted) array
// Output: A modified to represent a heap.
// Running Time: $O(n)$ where $n = \text{length}[A]$
1 $\text{heap-size}[A] \leftarrow \text{length}[A]$
2 **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

HEAP-INCREASE-KEY(A, i, key)

// Input: A : an array representing a heap, i : an array index, key : a new key greater than $A[i]$

// Output: A still representing a heap where the key of $A[i]$ was increased to key

// Running Time: $O(\log n)$ where $n = \text{heap-size}[A]$

```
1 if  $key < A[i]$ 
2   error("New key must be larger than current key")
3  $A[i] \leftarrow key$ 
4 while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5   exchange  $A[i]$  and  $A[\text{PARENT}(i)]$ 
6    $i \leftarrow \text{PARENT}(i)$ 
```

HEAP-SORT(A)

// Input: A : an (unsorted) array

// Output: A modified to be sorted from smallest to largest

// Running Time: $O(n \log n)$ where $n = \text{length}[A]$

```
1 BUILD-MAX-HEAP( $A$ )
2 for  $i = \text{length}[A]$  downto 2
3   exchange  $A[1]$  and  $A[i]$ 
4    $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
5   MAX-HEAPIFY( $A, 1$ )
```

HEAP-EXTRACT-MAX(A)

// Input: A : an array representing a heap

// Output: The maximum element of A and A as a heap with this element removed

// Running Time: $O(\log n)$ where $n = \text{heap-size}[A]$

```
1  $max \leftarrow A[1]$ 
2  $A[1] \leftarrow A[\text{heap-size}[A]]$ 
3  $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
4 MAX-HEAPIFY( $A, 1$ )
5 return  $max$ 
```

MAX-HEAP-INSERT(A, key)

// Input: A : an array representing a heap, key : a key to insert

// Output: A modified to include key

// Running Time: $O(\log n)$ where $n = \text{heap-size}[A]$

```
1  $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$ 
2  $A[\text{heap-size}[A]] \leftarrow -\infty$ 
3 HEAP-INCREASE-KEY( $A[\text{heap-size}[A]], key$ )
```

1 Overview

- Overview of Heaps
- Heap Algorithms (Group Exercise)
- More Heap Algorithms!
- Master Theorem Review

2 Heap Overview

Things we can do with heaps are:

- insert
- max
- extract_max
- increase_key
- build them
- sort with them

(Max-)Heap Property For any node, the keys of its children are less than or equal to its key.

3 Heap Algorithms (Group Exercise)

We split into three groups and took 5 or 10 minutes to talk. Then each group had to work their example algorithm on the board.

Group 1: MAX-HEAPIFY and BUILD-MAX-HEAP

Given the array in Figure 1, demonstrate how BUILD-MAX-HEAP turns it into a heap. As you do so, make sure you explain:

- How you visualize the array as a tree (look at the PARENT and CHILD routines).
- The MAX-HEAPIFY procedure and why it is $O(\log(n))$ time.
- That early calls to MAX-HEAPIFY take less time than later calls.

The correct heap is also shown in Figure 1.

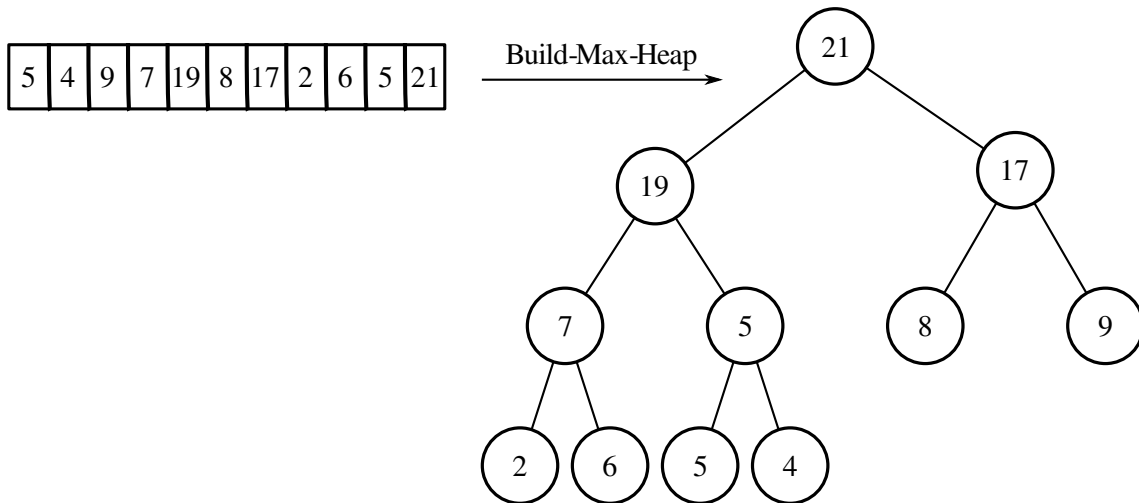


Figure 1: The array to sort and the heap you should find.

Group 2: HEAP-INCREASE-KEY

For the heap shown in Figure 2 (which Group 1 will build), show what happens when you use HEAP-INCREASE-KEY to increase key 2 to 22. Make sure you argue why what you're doing is $O(\log n)$. (Hint: Argue about how much work you do at each level)

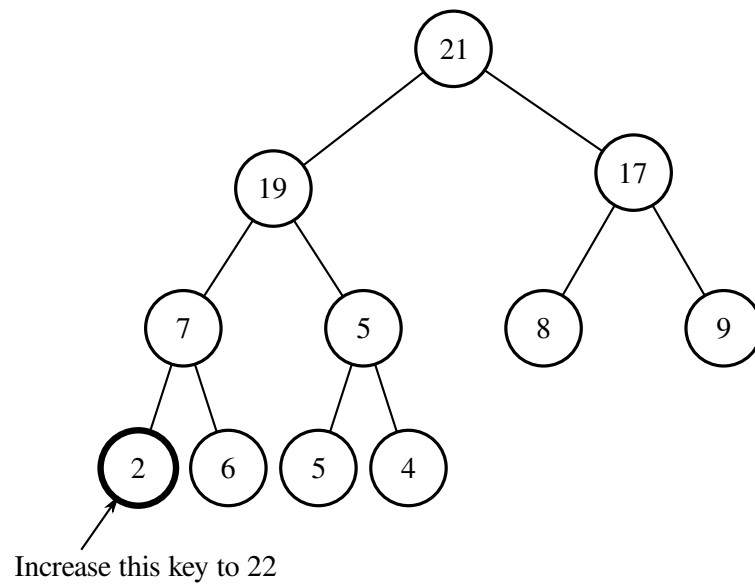


Figure 2: The heap on which to increase a key. You should increase the key of the bottom left node (2) to be 22.

Group 3: HEAP-SORT

Given the heap shown in Figure 3 (which Groups 1 and 2 will build for you), show how you use it to sort. You do not need to explain the MAX-HEAPIFY or the BUILD-MAX-HEAP routine, but you should make sure you explain why the runtime of this algorithm is $O(n \log n)$. Remember the running time of MAX-HEAPIFY is $O(\log n)$.

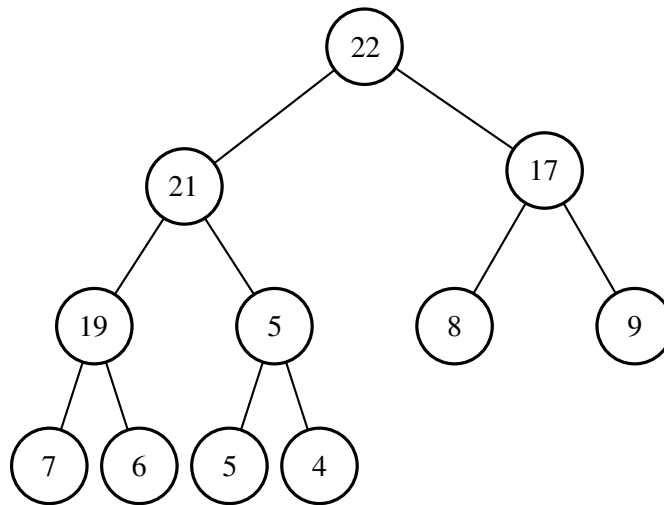


Figure 3: Sort this heap.

4 More Heap Algorithms

Note HEAP-EXTRACT-MAX and MAX-HEAP-INSERT procedures since we didn't discuss them in class:

HEAP-EXTRACT-MAX(A)

```
1   $max \leftarrow A[1]$ 
2   $A[1] \leftarrow A[heap-size[A]]$ 
3   $heap-size[A] \leftarrow heap-size[A] - 1$ 
4  MAX-HEAPIFY( $A, 1$ )
5  return  $max$ 
```

MAX-HEAP-INSERT(A, key)

```
1   $heap-size[A] \leftarrow heap-size[A] + 1$ 
2   $A[heap-size[A]] \leftarrow -\infty$ 
3  HEAP-INCREASE-KEY( $A[heap-size[A]], key$ )
```

5 Running Time of BUILD-MAX-HEAP

Trivial Analysis: Each call to MAX-HEAPIFY requires $\log(n)$ time, we make n such calls $\Rightarrow O(n \log n)$.

Tighter Bound: Each call to MAX-HEAPIFY requires time $O(h)$ where h is the height of node i . Therefore running time is

$$\begin{aligned} \sum_{h=0}^{\log n} \underbrace{\frac{n}{2^h + 1}}_{\text{Number of nodes at height } h} \times \underbrace{O(h)}_{\text{Running time for each node}} &= O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n) \end{aligned} \tag{1}$$

Note $\sum_{h=0}^{\infty} h/2^h = 2$.

6 Proving BUILD-MAX-HEAP Using Loop Invariants

(We didn't get to this in this week's recitation, maybe next time).

Loop Invariant: Each time through the **for** loop, each node greater than i is the root of a max-heap.

Initialization: At the first iteration, each node larger than i is at the root of a heap of size 1, which is trivially a heap.

Maintainance: Since the children of i are larger than i , by our loop invariant, the children of i are roots of max-heaps. Therefore, the requirement for MAX-HEAPIFY is satisfied and, at the end of the loop, index i also roots a heap. Since we decrement i by 1 each time, the invariant holds.

Termination: At termination, $i = 0$ so $i = 1$ is the root of a max-heap and therefore we have created a max-heap.

Discussion: What is the loop invariant for HEAP-SORT? (All keys greater than i are sorted).

Initialization: Trivial.

Maintainance: We always remove the largest value from the heap. We can call MAX-HEAPIFY because we have shrunk the size of the heap so that the root's children are root's of good heaps (although the root is not the root of a good heap).

Termination: $i = 0$

7 Master Theorem Review: More Examples

TRAVERSE-TREE(T)

```

1  if  $left-child(root[T]) == \text{NULL}$  and  $right-child(root[T]) == \text{NULL}$  return
2  output  $left-child(root[T]), right-child(root[T])$ 
3  TRAVERSE-TREE( $right-child(root[T])$ )
4  TRAVERSE-TREE( $left-child(root[T])$ )
```

Recurrence is $T = 2T(n/2) + O(1)$. $a = 2, b = 2, n^{\log_b(a)} = n, f(n) = 1$. Master Theorem Case 1, Running Time $O(1)$.

MULTIPLY(x, y)

```

1   $n \leftarrow \max(|x|, |y|)$  //  $|x|$  is size of  $x$  in bits
2  if  $n = 1$  return  $xy$ 
3   $x_L \leftarrow x[1 : n/2], x_R \leftarrow x[n/2 + 1 : n], y_L \leftarrow y[1 : n/2], y_R \leftarrow y[n/2 + 1 : n]$ 
4   $P_1 = \text{MULTIPLY}(x_L, y_L)$ 
5   $P_2 = \text{MULTIPLY}(x_R, y_R)$ 
6   $P_3 = \text{MULTIPLY}(x_L + x_R, y_L + y_R)$ 
7  return  $2^n P_1 + 2^{n/2} (P_3 - P_1 - P_2) + P_2$ 
```

Recurrence Relation: $T(n) = 3T(n/2) + O(n)$ (Note: Addition takes linear time in number of bits). $a = 3, b = 2, n^{\log_b(a)} = n^{\log_2(3)}, f(n) = O(n)$, Case 1 of Master Theorem, $O(n^{\log_2(3)})$

MATRIXMULTIPLY(X, Y)

```
1   $n \leftarrow \text{sizeof}(X)$  // Assume  $X$  and  $Y$  are the same size and square
2  if  $n = 1$ , return  $XY$ 
3  // Split  $X$  and  $Y$  into four quadrants:
    $A \leftarrow \text{UpperLeft}(X)$ ,  $B \leftarrow \text{UpperRight}(X)$ ,  $C \leftarrow \text{LowerLeft}(X)$ ,  $D \leftarrow \text{LowerRight}(X)$ 
    $E \leftarrow \text{UpperLeft}(Y)$ ,  $F \leftarrow \text{UpperRight}(Y)$ ,  $G \leftarrow \text{LowerLeft}(Y)$ ,  $H \leftarrow \text{LowerRight}(Y)$ 
4   $UL \leftarrow \text{MATRIXMULTIPLY}(A, E) + \text{MATRIXMULTIPLY}(B, G)$ 
5   $UR \leftarrow \text{MATRIXMULTIPLY}(A, F) + \text{MATRIXMULTIPLY}(B, H)$ 
6   $LL \leftarrow \text{MATRIXMULTIPLY}(C, E) + \text{MATRIXMULTIPLY}(D, G)$ 
7   $LR \leftarrow \text{MATRIXMULTIPLY}(C, F) + \text{MATRIXMULTIPLY}(D, H)$ 
8  return matrix with  $UL$  as upper left quadrant,  $UR$  as upper right,  $LL$  as lower left,  $LR$  as lower right.
```

Recurrence Relation: $T(n) = 8T(n/2) + O(n^2)$. $a = 8, b = 2, n^{\log_b(a)} = n^3, f(n) = n^2$. Case 1 of the Master Theorem, $O(n^3)$.