CMPU241 Analysis of Algorithms

Heapsort (chapter 6)

Sorting Algorithm Terminology

- A sorting algorithm is *comparison-based* if the only operation we can perform on keys is to compare them.
- A sorting algorithm is *in place* if only a constant number of elements of the input array are ever stored outside the array.

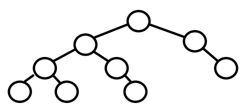
Running Time of Comparison-Based Sorting Algorithms

	worst-case	average-case	best-case	in place?
Insertion Sort	n^2	n^2	n	yes
Merge Sort	$n\lg n$	$n\lg n$	$n\lg n$	no
Heap Sort				
Quick Sort				

Binary Trees

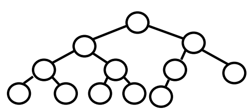
binary tree

- A rooted tree in which each internal node has at most 2 children



complete binary tree

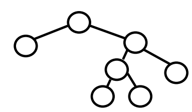
- Each level of the binary tree is full except possibly the lowest. All nodes on lowest level are as far left as possible (ie, tree is left filled).



Binary Trees

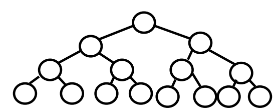
full binary tree

- A binary tree in which every internal node has 2 children (aka proper binary tree).

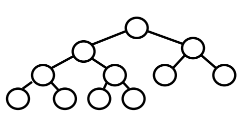


full and complete binary tree

- Each level of the binary tree is full.



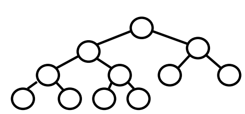
Binary Heaps



heap

- A data structure used to efficiently find the largest or smallest element in a set.
- Each node contains a key and the keys are some totally ordered, comparable type of data.

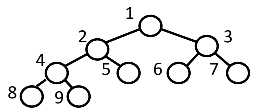
Binary Heaps



binary heap

- A binary heap is a complete binary tree, i.e., it may be missing some rightmost leaves on the bottom level. *The bottom level is left-filled.*

Binary Heaps



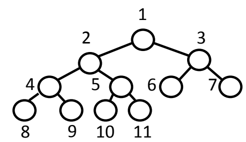
Heap implementation is an array-embedded binary tree

- Encoding stores tree elements at particular indexes in an array.
- Uses "level-ordering" and 1-based indexing (our textbook).

Heapsort

In an array-embedded implementation of a heap:

- ***heapsize*** is number of elements in heap
- ***length*** is number of positions in array
  - **Invariant: *length* ≥ *heapsize*.**



Max-Heap

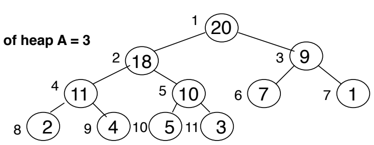
In the 1-based array representation of a max-heap, the root of the tree is in A[1], and given the index i of a node,

Parent(i)	LeftChild(i)	RightChild(i)
return (⌊i/2⌋)	return (2i)	return (2i + 1)

Max-heap invariant: A[Parent(i)] > A[i]

	1	2	3	4	5	6	7	8	9	10	11	: index
A	20	18	9	11	10	7	1	2	4	5	3	: keys

n = 11
height of heap A = 3



Heap-sort is accomplished with a max-heap

0-Based Max-Heap

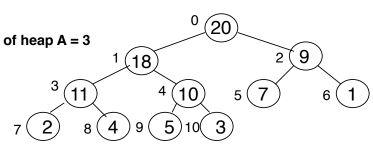
In the 0-based array representation of a max-heap, the root of the tree is in A[0], and given the index i of a node,

Parent(i)	LeftChild(i)	RightChild(i)
return (⌊(i -1)/2⌋)	return (2i +1)	return (2i + 2)

Max-heap invariant: A[Parent(i)] > A[i]

	0	1	2	3	4	5	6	7	8	9	10	: index
A	20	18	9	11	10	7	1	2	4	5	3	: keys

n = 11
height of heap A = 3



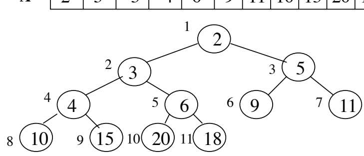
Min-Heap

In the 1-based array representation of a min-heap, the root of the tree is in A[1], and given the index i of a node,

Parent(i)	LeftChild(i)	RightChild(i)
return (⌊i/2⌋)	return (2i)	return (2i + 1)

Min-heap invariant: A[Parent(i)] < A[i]

	1	2	3	4	5	6	7	8	9	10	11	: index
A	2	3	5	4	6	9	11	10	15	20	18	: keys



Min-heaps are used, e.g., as priority queues in event-driven simulators.

Creating a Heap: Build-Max-Heap

- Starting from an unordered array of n elements.
- Observation: Leaves are already trivial max-heaps.
  Elements A[(⌊n/2⌋ + 1) ... n] are all leaves.
- Start at parents of leaves...then go to grandparents of leaves...moving larger values up the tree and moving lower values down in the tree.

```
Build-Max-Heap(A)
1.  for i = ⌊A.length/2⌋ downto 1
2.       Max-Heapify(A, i)
```

Running Time of Build-Max-Heap

- About *n/2* calls to Max-Heapify (O(n) calls)

Max-Heapify: Maintaining the Max-Heap Property

Precondition: the subtrees rooted at 2i and 2i+1 are max-heaps

```
Max-Heapify(A, i)   /* Max-Heapify is also known as "Sink" */
left = 2i ; right = 2i + 1
largest = i         /* set largest to i, parent node of left and right */
if left ≤ A.heapsize and A[left] > A[i]
    largest = left  /* reset largest to left child
if right ≤ A.heapsize and A[right] > A[largest]
    largest = right /* reset largest to right child */
if largest != i     /* keep sinking i in tree */
    swap(A[i], A[largest])
    Max-Heapify(A, largest)  /* continue heapifying toward leaves */
```

Sink: Alternate version of Max-Heapify

Create variables with global scope:

```
int heapsize;
E[] array = (E[]) new Object[length];
```

Sink compares (possibly smaller) parent i to (possibly larger) left and right children and swaps key of child with largest key with parent. Correctness relies on the precondition that the left and right children are the roots of max-heaps.

Assume size >> heapsize and heapsize is highest-numbered node in heap.

Sink: Iterative version of Max-Heapify

Precondition: the subtrees 2i and 2i+1, are max-heaps. Let n = heapsize

```
private void sink(int k)
{
  while (2*k <= n)
  {
    int j = 2*k;
    if ( j < n && less( j, j+1)) j++;
    if ( !less( k, j )) break;
    swap( k, j );
    k = j;
  }
}
```

```
private boolean less(int i, int j)
{
   return A[i].compareTo(A[j]) < 0;
}
```

```
private void swap(int i, int j)
{
   int t = A[i];
   A[i] = A[j];
   A[j] = t;
}
```

Max-Heapify: Maintaining the Max-Heap Property

- *Precondition*: subtrees rooted at the left and right children of A[i], A[2i] and A[2i + 1] are max-heaps (i.e., they obey the max-heap property)

  ...but subtree rooted at A[i] might not be a max-heap (that is, A[i] may be smaller than its left and/or right child)

- *Postcondition:* Max-Heapify will cause the value at A[i] to "float down" or "sink" in the heap until the subtree rooted at A[i] becomes a heap.

In a totally unordered array, execution would start at the highest numbered parent node of a leaf.

Max-Heapify: Running Time

Running Time of Max-Heapify

- every line is $\theta(1)$ time except the recursive call
- in worst-case, last level of binary tree is half empty, so the sub-tree rooted at left child has size < $(2/3)n$

We get the recurrence $T(n) \leq T(2n/3) + \theta(1)$

which, by case 2 of the master theorem, has the solution

$$T(n) = O(\lg n)$$

(or, Max-Heapify takes O(h) time when node A[i] has height h in the heap) The height h of the tree is the root to leaf path with the most edges. $O(h) = O(\lg n)$

Build-Max-Heap - Tighter bound

Proposition 1:

Sink-based heap construction uses fewer than 2n compares and fewer than n exchanges to construct a heap from n items.

Proof sketch:

Follows from the observation that most of the heaps processed are small. E.g., to build a heap of 127 items, we process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63, and 1 heap of size 127. Thus, there are 32X1 + 16X2 + 8X3 + 4X4 + 2X5 + 1X6 = 120 exchanges (and twice as many compares) required (at worst).

Inserting Heap Elements

Inserting an element into a max-heap:
- increment heapsize and "add" new element to the highest numbered position of array
- walk up tree from new leaf to root, swapping values. Insert input key at node in which a parent key larger than the input key is found

```
Max-Heap-Insert(A, key)
1. A.heapsize = A.heapsize +1
2. i = A.heapsize
3. while i > 1 and A[parent(i)] < key
4.      A[i] = A[parent(i)]
5.      i = parent(i)
6. A[i] = key
```

Here, values are floated up to where they should be in a max-heap.

Precondition: A is a max-heap

Running time of Max-Heap-Insert: O(lgn)
- time to traverse leaf to root path (height = O(lgn))

Swim: another version of Max-Heap-Insert

Precondition: the key is inserted into what is initially a max-heap at the position A[heapsize++] and then swim is called with k = heapsize.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   // k != root and k > parent
   {
      swap( k, k/2 );
      k = k/2; // set k to parent
   }
}
```

```
private boolean less(int i, int j)
{
   return A[i].compareTo(A[j]) < 0;
}
```

```
private void swap(int i, int j)
{
   int t = A[i];
   A[i] = A[j];
   A[j] = t;
}
```

This algorithm uses a 1-based array

Correctness of Build-Max-Heap

Loop invariant: At the start of each iteration *i* of the for loop, each node *i* + 1, *i* + 2, ..., *n* is the root of a max-heap.

- Initialization: $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots n$ is a leaf, trivially satisfying the max-heap property.
- Inductive hypothesis: At the start of iteration k ($1 \le k \le \lfloor n/2 \rfloor$), the subtrees of *k* are the roots of max-heaps
- Inductive step (maintenance): During iteration k, Max-Heapify is called on node k. By the IH, the left and right subtrees of k are max-heaps. When Max-Heapify is called on node k, the value in node k is "floated down" in its subtree until its value is correctly positioned in the max-heap rooted at k.

```
Build-Max-Heap(A)
1. A.heapsize = A.length
2. for i = ⌊A.length/2⌋ downto 1
3.     Max-Heapify(A, i)
```

Correctness of Build-Max-Heap

Termination: at termination, *i = 0*. By the loop invariant, nodes *1, 2, ...,n* are the roots of max-heaps. Therefore the algorithm is correct because it produces a max-heap.

```
Build-Max-Heap(A)
1. A.heapsize = A.length
2. for i = ⌊A.length/2⌋ downto 1
3.     Max-Heapify(A, i)
```

Loop invariant: At the start of each iteration *i* of the for loop, each *node i + 1, i + 2, ..., n* is the root of a max-heap.

Heap Sort

Input: An *n*-element array A (unsorted).
Output: An *n*-element array A in sorted order, smallest to largest.

```
HeapSort(A)
1. Build-Max-Heap(A)   /* rearrange elements to form max heap */
2. for i = A.length downto 2 do
3.      swap A[1] and A[i]      /* puts max in ith array position */
4.      A.heapSize = A.heapSize – 1      /* decrease heap size */
5.      Max-Heapify(A,1) /* restore heap property from node 1 */
```

Build-Max-Heap(A) takes O(n) time

Max-Heapify(A,1) takes O(lgn) time

Running time of HeapSort
- 1 call to Build-Max-Heap() ⇒ O(n) time
- n-1 calls to Max-Heapify() each takes O(lgn) time ⇒ O(nlgn) time

Iterative version of Heap Sort

Input: An *n*-element array A (unsorted).
Output: An *n*-element array A in sorted order, smallest to largest.

```
public static void sort(Comparable[] A)
{
   int n = A.length ;
   // start at highest numbered parent node
   for (int k = n/2; k >= 1; k--)
       sink(A, k, n);
   while (n > 1)
   {
      swap(A, 1, n--);
      sink(A, 1, n);
   }
}
```

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

Iterative version of Heap Sort

Input: An *n*-element array A (unsorted).
Output: An *n*-element array A in sorted order, smallest to largest.

```
public static void sort(Comparable[] A)
{
  int n = A.length ;
  // start at highest numbered parent node
  for (int k = n/2; k >= 1; k--)
      sink(A, k, n);
  while (n > 1)
  {
    swap(A, 1, n--);
    sink(A, 1, n);
  }
}
```

Running time of sort

- for loop iterates n/2 times. calls sink on every iteration
  ⇒ O(nlgn) time
- while loop has n-1 calls to sink() each takes O(lgn) time
  ⇒ O(nlgn) time

Heapsort Time and Space Usage

- An array implementation of a heap uses *O(n)* space
  -one array element for each node in heap
- Heapsort uses *O(n)* space and is **in place**, meaning at most constant extra space beyond that taken by the input is needed
- Running time is as good as merge sort, O(nlgn) in worst case.

Heaps as Priority Queues

Definition: A **priority queue** is a data structure for maintaining a set S of elements, each with an associated key. A max-heap gives priority to keys with larger values and supports the following operations:

1. insert(A, x) inserts the element x into array at next highest position in A.
2. max(A) returns value of element A with largest key.
3. extract-max(A) removes and returns element of A with largest key.
4. increase-key(A,x,k) increases the value of element x's key to new value k (assuming k is at least as large as current key's value).

Priority Queues

An application of max-priority queues is to schedule jobs on a shared processor. Need to be able to

check current job's priority	**Heap-Maximum(A)**
remove job from the queue	**Heap-Extract-Max(A)**
insert new jobs into queue	**Max-Heap-Insert(A, key)**
increase priority of jobs	**Heap-Increase-Key(A,i,key)**

Initialize PQ by running Build-Max-Heap on an array A.

A[1] holds the maximum value after this step.

Heap-Maximum(A) - returns value of A[1] (does nothing to heap).
Heap-Extract-Max(A) - Saves A[1] and then, like Heap-Sort, puts item in A[heapsize] at A[1], decrements heapsize, and uses Max-Heapify(A, 1) to restore heap property.

Heap-Increase-Key

Heap-Increase-Key(A, i, key) - If key is larger than current key at A[i], moves new value in A[i] up heap until heap property is restored.

An application for a min-heap priority queue is an event-driven simulator, where the key is an integer representing the number of seconds (or other discrete time unit) from time zero (starting point for simulation).

Sorting Algorithms

- A sorting algorithm is *comparison-based* if the only operation we can perform on keys is to compare them.
- A sorting algorithm is *in place* if only a constant number of elements of the input array are ever stored outside the array.

Running Time of Comparison-Based Sorting Algorithms

	worst-case	**average-case**	**best-case**	**in place?**
Insertion Sort	n^2	n^2	n	yes
Merge Sort	$n\lg n$	$n\lg n$	$n\lg n$	no
Heap Sort	$n\lg n$	$n\lg n$	$n\lg n$	yes
Quick Sort				

Build-Max-Heap - Tighter bound

Build-Max-Heap(A)
1. A.heapsize = A.length
2. for ***i*** ← ⌊***length***(A)/2⌋ downto **1**
3. Max-Heapify(A, i)

Proof of tighter bound (O(n)) relies on following theorem:

Theorem 1: The number of nodes at height h in a max-heap ≤ $\lceil n/2^{h+1} \rceil$

Height of a node v = longest distance from v to a leaf.
Depth of a node v = distance from node v to the root.

Tight analysis relies on the properties that an n-node heap has height floor of lgn and at most the ceiling of $n/2^{h+1}$ nodes at height h. The time for max-heapify to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

Lemma 1: The number of internal nodes in a proper binary tree is equal to the number of leaves in the tree - 1.

Defn: In a proper binary tree (pbt), each node has exactly 0 or 2 children.

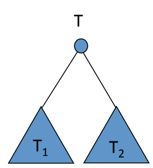
Let I be the number of internal nodes and let L be the number of leaves in a proper binary tree T. The proof is by induction on the height of T.

Basis: h=0. I = 0 and L = 1. I = L - 1 = 1 - 1 = 0, so the lemma holds.

Inductive Step: Assume lemma is true for proper binary trees of height h (IHOP) and show for proper binary trees of height h + 1.

Consider the root of a proper binary tree T of height h+1. It has left and right subtrees (L and R) of height at most h.
$I_T = (I_L + I_R) + 1 = (L_L - 1) + (L_R - 1) + 1$ (by the IHOP) =
$(L_L + L_R - 2) + 1 = L_L + L_R - 1$. Since $L_T = L_L + L_R$ we have that $I_T = L_T - 1$.
QED

Diagrammatic proof of Lemma 1



#Internal nodes in T	=	#Internal nodes in T_1 + #Internal nodes in T_2 + 1
	=	(#Leaves in T_1 - 1) + (#Leaves in T_2 -1) + 1 (IHOP)
	=	(#Leaves in T_1 + #Leaves in T_2) - 2 + 1
	=	#Leaves in T - 1 (by observation that # of leaves in T is equal to # leaves in its subtrees.)

Theorem 1: The number of nodes at level h in a max-heap ≤ $\lceil n/2^{h+1} \rceil$

Let H be the height of the heap. Proof is by induction on h, the height of each node. The number of nodes in the heap is n.

Basis: Show the theorem holds for nodes with h = 0. The tree leaves (nodes at height 0) are at depth H.

Let x be the number of nodes at depth H, that is, the number of leaves, assuming that the tree is a complete binary tree, i.e., that $n = 2^{H+1} - 1$

Note that n - x is odd, because a complete binary tree has an odd number of internal nodes (1 less than a power of 2) and an even number of leaves = 2^H

Theorem 1: The number of nodes at level h in a max-heap ≤ $\lceil n/2^{h+1} \rceil$...(basis continued)

We have that n is odd and x is even, so all leaves have siblings (all internal nodes have 2 children.) By Lemma 1, the number of internal nodes = the number of leaves - 1.

So n = # of nodes = # of leaves + # internal nodes = 2(# of leaves) - 1. Thus, the # of leaves = (n+1)/2 = $\lceil n/2^{0+1} \rceil$ because n is odd.

Thus, the number of leaves = $\lceil n/2^{0+1} \rceil$ and the theorem holds for the base case.

Theorem 1: The number of nodes at level h in a max-heap ≤ $\lceil n/2^{h+1} \rceil$

Inductive step: Show that if thm 1 holds for height h-1, it holds for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T′ formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

Note that each node at height h (e.g. 1) in T would be at height h-1 (e.g. 0) if the leaves of the tree were removed--i.e., they are at height h-1 in T′. Letting n'_{h-1} denote the number of nodes at height h-1 in T′, we have $n_h = n'_{h-1}$

$n_h = n'_{h-1} \le \lceil n'/2^h \rceil$ (by the IHOP) $= \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$.

Since the time of Max-Heapify when called on a node of height h is O(h), the time of B-M-H is

$$\sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\lg n} \frac{h}{2^h})$$

and since the last summation turns out to be a constant, the running time is O(n).

Therefore, we can build a max-heap from an unordered array in linear time.