Algorithms

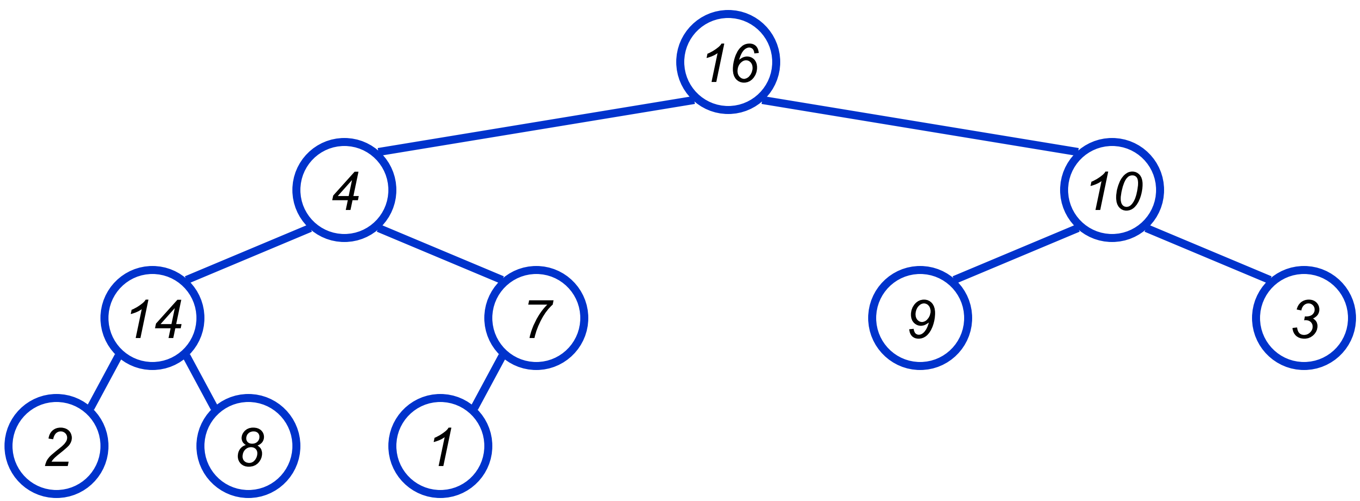
Heapsort

Priority Queues

Quicksort

Review: Heaps

- A *heap* is a "complete" binary tree, usually represented as an array:



A =	16	14	10	8	7	9	3	2	4	1

Review: Heaps

- To represent a heap as an array:

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

Review: The Heap Property

- Heaps also satisfy the *heap property*:

  $A[\mathit{Parent}(i)] \geq A[i]$ for all nodes $i > 1$

  - In other words, the value of a node is at most the value of its parent
  - The largest value is thus stored at the root (A[1])
- Because the heap is a binary tree, the height of any node is at most $\Theta(\lg n)$

Review: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *l* and *r*
  - Given: two subtrees rooted at *l* and *r*, assumed to be heaps
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - If A[i] < A[l] or A[i] < A[r], swap A[i] with the largest of A[l] and A[r]
    - Recurse on that subtree
  - Running time: O(h), h = height of heap = O(lg n)

Review: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)
  - So:
    - Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
    - Order of processing guarantees that the children of node *i* are heaps when *i* is processed

BuildHeap()

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
     Heapify(A, i);
}
```

Analyzing BuildHeap()

- Each call to **`Heapify()`** takes $O(\lg n)$ time
- There are $O(n)$ such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is $O(n \lg n)$
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is $O(n)$
  - *How can this be? Is there a flaw in the above reasoning?*

Analyzing BuildHeap(): Tight

- To **`Heapify()`** a subtree takes O(h) time where h is the height of the subtree
  - $h = O(\lg m)$, m = # nodes in subtree
  - The height of most subtrees is small
- Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- CLR 7.3 uses this fact to prove that **`BuildHeap()`** takes O(n) time

Heapsort

- Given `BuildHeap()`, an in-place sorting algorithm is easily constructed:
  - Maximum element is at A[1]
  - Discard by swapping with element at A[n]
    - Decrement heap_size[A]
    - A[n] now contains correct value
  - Restore heap property at A[1] by calling `Heapify()`
  - Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
      BuildHeap(A);
      for (i = length(A) downto 2)
      {
            Swap(A[1], A[i]);
            heap_size(A) -= 1;
            Heapify(A, 1);
      }
}
```

Analyzing Heapsort

- The call to **`BuildHeap()`** takes $O(n)$ time
- Each of the $n - 1$ calls to **`Heapify()`** takes $O(\lg n)$ time
- Thus the total time taken by **`HeapSort()`**
  $= O(n) + (n - 1) O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**
  - ***What might a priority queue be useful for?***

Priority Queue Operations

- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key
- *How could we implement these operations using a heap?*

Tying It Into The Real World

- And now, a real-world example…

Tying It Into The "Real World"

- And now, a real-world example…*combat billiards*
  - Sort of like pool...
  - Except you're trying to kill the other players…
  - And the table is the size of a polo field…
  - And the balls are the size of Suburbans...
  - And instead of a cue you drive a vehicle with a ram on it

Figure 1: boring traditional pool

- Problem: *how do you simulate the physics?*

Combat Billiards: Simulating The Physics

- Simplifying assumptions:
  - G-rated version: No players
    - Just *n* balls bouncing around
  - No spin, no friction
    - Easy to calculate the positions of the balls at time T_n from time T_{n-1} if there are no collisions in between
  - Simple elastic collisions

Simulating The Physics

- Assume we know how to compute when two moving spheres will intersect
  - Given the state of the system, we can calculate when the next collision will occur for each ball
  - At each collision C_i:
    - Advance the system to the time T_i of the collision
    - Recompute the next collision for the ball(s) involved
    - Find the next overall collision C_{i+1} and repeat
  - *How should we keep track of all these collisions and when they occur?*

Implementing Priority Queues

```
HeapInsert(A, key)     // what's running time?
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```

Implementing Priority Queues

```
HeapMaximum(A)
{
    // This one is really tricky:

    return A[i];
}
```

Implementing Priority Queues

```
HeapExtractMax(A)
{
    if (heap_size[A] < 1) { error; }
    max = A[1];
    A[1] = A[heap_size[A]]
    heap_size[A] --;
    Heapify(A, 1);
    return max;
}
```

Back To Combat Billiards

- Extract the next collision C_i from the queue
- Advance the system to the time T_i of the collision
- Recompute the next collision(s) for the ball(s) involved
- Insert collision(s) into the queue, using the time of occurrence as the key
- Find the next overall collision C_{i+1} and repeat

Using A Priority Queue For Event Simulation

- More natural to use **Minimum()** and **ExtractMin()**
- *What if a player hits a ball?*
  - Need to code up a **Delete()** operation
  - *How? What will the running time be?*

Quicksort

- Sorts in place
- Sorts $O(n \lg n)$ in the average case
- Sorts $O(n^2)$ in the worst case
- *So why would people use it instead of merge sort?*

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How do you suppose we implement this function?*

Partition In Words

- Partition(A, p, r):
  - Select an element to act as the "pivot" (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

Illustrate on
A = {5, 3, 2, 6, 4, 1, 3, 7};

What is the running time of **`partition()`***?*