CS 361, Lecture 14

Jared Saia
University of New Mexico

Administrative

- Midterm: Next Thursday in class. Will contain at least one problem like problems 1-3, will contain one problem like problem 4, will contain one problem on loop invariants (similar to problem 5). Will also contain other types of problems!
- You can bring 2 pages of "cheat sheets" to use during the exam. Otherwise the exam is closed book and closed note
- Note that the web page contains links to prior classes and their midterms. *Many of the questions on my midterm will be similar in flavor to these past midterms (and to exercises in the book)!*

Admistrative

- Note on problem 4 on hw: similar to midterm problem from Cris Moore's class (solutions are online). Also: Recall that $k^n = O(k^n)$ for any constant k

Outline

"For NASA, space is still a high priority", Dan Quayle

- Priority Queues

Heap-Sort Review

Heap-Sort (A)

1. Build-Max-Heap (A)
2. for (i=length (A);$i > 1$;$i--$)
   (a) do exchange A[1] and A[i]
   (b) heap-size (A) = heap-size (A) - 1
   (c) Max-Heapify (A,1)

Analysis

- Heap-Sort takes $O(n \log n)$ time. Q: What is best case run-time? Q: What is runtime if the array is already in sorted order?
- Q: Correctness?

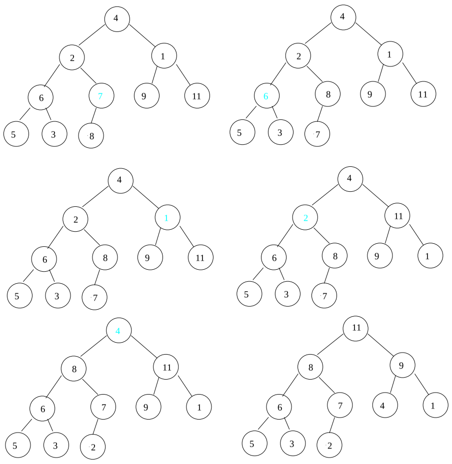
Build-Max-Heap

Build-Max-Heap (A)

1. heap-size (A) = length (A)
2. for ($i = \lfloor length(A)/2 \rfloor$;$i > 0$;$i - -$)
   (a) do Max-Heapify (A,i)

Example

A=4 2 1 6 7 9 11 5 3 8

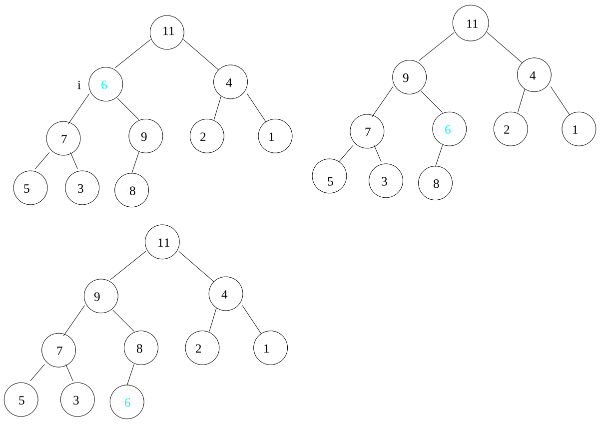


Max-Heapify

Max-Heapify (A,i)

1. $l = Left(i)$
2. $r = Right(i)$
3. $largest = i$
4. if ($l \leq$ heap-size(A) and $A[l] > A[i]$) then $largest = l$
5. if ($r \leq$ heap-size(A) and $A[r] > A[largest]$) then $largest = r$
6. if $largest \neq i$ then
   (a) exchange A[i] and A[largest]
   (b) Max-Heapify (A,largest)

Example



Priority Queues

A Priority Queue is an ADT for a set S which supports the following operations:

- *Insert (S,x)*: inserts x into the set S
- *Maximum (S)*: returns the maximum element in S
- *Extract-Max (S)*: removes and returns the element of S with the largest key
- *Increase-Key (S,x,k)*: increases the value of x's key to the new value k (k is assumed to be as large as x's current key)

(note: can also have an analagous min-priority queue)

Applications of Priority Queue

- Application: Scheduling jobs on a workstation
- Priority Queue holds jobs to be performed and their priorities
- When a job is finished or interrupted, highest-priority job is chosen using Extract-Max
- New jobs can be added using Insert

(note: an application of a min-priority queue is scheduling events in a simulation)

Implementation

- A Priority Queue can be implemented using heaps
- We'll show how to implement each of these four functions using heaps

Heap-Maximum

Heap-Maximum (A)

1. return A[1]

Heap-Extract-Max

Heap-Extract-Max (A)

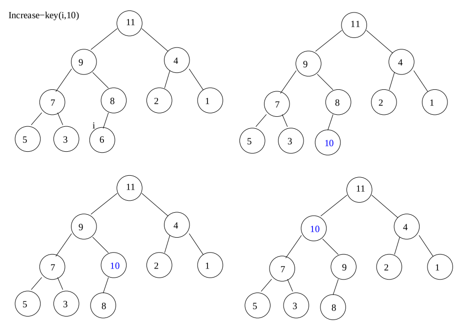
1. if (heap-size (A)<1) then return "error"
2. max = A[1];
3. A[1] = A[heap-size (A)];
4. heap-size (A)−−;
5. Max-Heapify (A,1);
6. return max;

Heap-Increase-Key

Heap-Increase-Key (A,i,key)

1. if (key < A[i]) then error "new key is smaller than current key"
2. A[i] = key;
3. while (i>1 and A[Parent (i)] < A[i])
   (a) do exchange A[i] and A[Parent (i)]
   (b) i = Parent (i);

Example



Heap-Insert

Heap-Insert (A,key)

1. heap-size (A) ++;
2. A[heap-size (A)] = - infinity
3. Heap-Increase-Key (A,heap-size (A), key)

Analysis

- Heap-Maximum takes $O(1)$ time
- Heap-Extract-Max takes $O(\log n)$
- Heap-Increase-Key takes $O(\log n)$
- Heap-Insert takes $O(\log n)$

Correctness?

In-Class Exercise

- Imagine you have a min-heap with the following operations defined and taking $O(\log n)$:
  - (key,data) Heap-Extract-Min (A)
  - Heap-Insert (A,key,data)
- Now assume you're given k sorted lists, each of length n/k
- Use this min-heap to give a $O(n \log k)$ algorithm for merging these k lists into one sorted list of size n.

In-Class Exercise

- Q1: What is the high level idea for solving this problem?
- Q2: What is the pseudocode for solving the problem?
- Q3: What is the runtime analysis?
- Q4: What would be an appropriate loop invariant for proving correctness of the algorithm?

In-Class Exercise

```
KMerge (int arrList[][], int n, int k)\{
int arrI[] = new int[k];
int arrRes[] = new int[n];
for (i=1;i<= k;i++){
  Heap-Insert (A,arrList[i][1],i);
  arrI[i] = 1;
}
for (i=1;i<=n;i++){
  (key,listNum) = Heap-Extract-Min (A);

   arrRes[i] = key;
   arrI[listNum]++;
   if (arrI[lisNum] <= n/k){
     Heap-Insert (A,arrList[listNum][arrI[listNum]], arrI[listNum]);
}
```


Quicksort

- Based on divide and conquer strategy
- Worst case is $\Theta(n^2)$
- Expected running time is $\Theta(n \log n)$
- An In-place sorting algorithm
- Almost always the fastest sorting algorithm

Quicksort

- **Divide:** Pick some element A[q] of the array A and partition A into two arrays A_1 and A_2 such that every element in A_1 is $\leq$ A[q], and every element in A_2 is $>$ A[p]
- **Conquer:** Recursively sort A_1 and A_2
- **Combine:** A_1 concatenated with $A[q]$ concatenated with A_2 is now the sorted version of A

Todo

- Finish Chapter 6
- Start Chapter 7