CS 361, Lecture 9

Jared Saia
University of New Mexico

Outline

“For NASA, space is still a high priority”, Dan Quayle

- Heap Sort
- Priority Queues
- Quicksort

Build-Max-Heap

Build-Max-Heap (A)

1. heap-size (A) = length (A)
2. for (i = [length(A)/2]; i > 0; i --)
   (a) do Max-Heapify (A,i)

Example
Max-Heapify (A,i)

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

Example

Heap-Sort Review

Heap-Sort (A)

1. Build-Max-Heap (A)
2. for (i=\text{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 runtime? Q: What is runtime if the array is already in sorted order?
- Q: Correctness?
Analysis

We can prove correctness by using the following loop invariant:

- At the start of each iteration of the for loop, the subarray $A[1..i]$ is a max-heap containing the $i$ smallest elements of $A[1..n]$ and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$ in sorted order.

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 analogous 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 (A)

1. return A[1]

Heap-Extract-Max (A)

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

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

Initially, the heap is represented as a binary tree:

```
12 9 7 11 10 8 5
```

After increasing key 4 to 10:

```
12 9 7 11 10 8 5
```

The modified heap is:

```
12 10 7 11 10 8 5
```
Heap-Insert

Heap-Insert (A, key)

1. heap-size (A) += 1;
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 $n$.

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?
A sorting algorithm is implemented as follows:

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

The Algorithm

```
//PRE: A is the array to be sorted, p>=1;
//     r is <= the size of A
//POST: A[p..r] is in sorted order
Quicksort (A,p,r){
    if (p<r){
        q = Partition (A,p,r);
        Quicksort (A,p,q-1);
        Quicksort (A,q+1,r);
    }
}
```

Partition

```
//PRE: A[p..r] is the array to be partitioned, p>=1 and r <= size
//     of A, A[r] is the pivot element
//POST: Let A' be the array A after the function is run. Then
//     A'[p..r] contains the same elements as A[p..r]. Further,
//     all elements in A'[p..res-1] are <= A[r], A'[res] = A[r],
//     and all elements in A'[res+1..r] are > A[r]
Partition (A,p,r){
    x = A[r];
    i = p-1;
    for (j=p;j<=r-1;j++){
        if (A[j]<=x){
            i++;
            exchange A[i] and A[j];
        }
    }
    exchange A[i+1] and A[r];
    return i+1;
}
```
Correctness of Partition

Basic idea: The array is partitioned into four regions, x is the pivot

- Region 1: Region that is less than or equal to x
- Region 2: Region that is greater than x
- Region 3: Unprocessed region
- Region 4: Region that contains x only

Region 1 and 2 are growing and Region 3 is shrinking

Example

- Consider the array (2 6 4 1 5 3)

Correctness

Basic idea: The array is partitioned into four regions, x is the pivot

- Region 1: Region that is less than or equal to x (between $p$ and $i$)
- Region 2: Region that is greater than x (between $i + 1$ and $j - 1$)
- Region 3: Unprocessed region (between $j$ and $r - 1$)
- Region 4: Region that contains x only ($r$)

Region 1 and 2 are growing and Region 3 is shrinking

Loop Invariant

At the beginning of each iteration of the for loop, for any index $k$:

1. If $p \leq k \leq i$ then $A[k] \leq x$
2. If $i + 1 \leq k \leq j - 1$ then $A[k] > x$
3. If $k = r$ then $A[k] = x$
Todo

- Finish Chapter 6
- Start Chapter 7