One Minute Review of Basic Data Structures

Dictionaries are ordered sets with

- **insert**
- **delete**
- **member**
- **min**
- **max**
- **predecessor**
- **successor**

These can be implemented by many data structures, including

- **unordered arrays**
- **ordered arrays**
- **linked lists**
- **balanced binary trees**
  - **red-black trees**
  - **b-trees**
  - **2-3 trees**
  - **...**

Implementations

Dictionaries are ordered sets with

- **insert**
- **delete**
- **member**
- **min**
- **max**
- **predecessor**
- **successor**

- **Balanced binary trees support all dictionary operations in $O(\log n)$ time.**
- **Arrays and lists support some operations in $O(n)$ and some in $O(1)$ depending on implementation.**
- **Hash tables can support insert, delete and member in expected $O(1)$ time.**

Priority Queue

We want to design a data structure that supports the following operations:

- **Insert**
- **Max**
- **ExtractMax**
- **IncreaseKey**

Balance binary trees can support all of these in $O(\lg n)$ time, but we would like something cleaner and simpler.

Heaps

Heap **is a binary tree that is**

- **filled in top down, left-to-right**
- **value of parent $\geq$ value of child**

Heap Representation

- **Number nodes from** 1
- $leftchild(i) = 2i$
- $rightchild(i) = 2i + 1$
- $parent(i) = \lfloor i/2 \rfloor$

Heap Operations

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Heap Operations

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

```
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        exchange A[i] with A[PARENT(i)]
6        i = PARENT(i)
```

MAX-HEAP-INSERT(A, key)

```
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

MaxHeapify

```
MAX-HEAPIFY(A, i)
1   l = LEFT(i)
2   r = RIGHT(i)
3   if l ≤ A.heap-size and A[l] > A[i]
4         largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7         largest = r
8   if largest ≠ i
9         exchange A[i] with A[largest]
10        MAX-HEAPIFY(A, largest)
```

First (correct) attempt at HeapSort

- **input is in** B
- **heap and output in** A

```
HEAPSORT(A)
1   for i = 1 to n
2         MAX-HEAP-INSERT(B[i])
3   for i = A.length downto 2
4         exchange A[1] with A[i]
5         A.heap-size = A.heap-size − 1
6         MAX-HEAPIFY(A, 1)
```

Analysis

- **2 sequential loops each** $O(n)$ **iterations**
- **heap operations take** $O(\lg n)$ **time**
- **total is** $O(n \lg n)$

Improvements?

- As we will see, $n \lg n$ is actually a lower bound that we can't beat
- Consider just the first loop, can we do it quicker?
- We are doing n heap inserts, BUT, we are doing them without any intervening operations, so it might be possible to do the sequence faster.

HeapSort

- **input starts unsorted in** A
- **heap and output in** A

HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

General Rules for Loop Invariant Proofs

We use loop invariants to help us understand why an algorithm is correct. We must show three things about a loop invariant:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

When the first two properties hold, the loop invariant is true prior to every iteration of the loop. Note the similarity to mathematical induction, where to prove that a property holds, you prove a base case and an inductive step. Here, showing that the invariant holds before the first iteration is like the base case, and showing that the invariant holds from iteration to iteration is like the inductive step.

The third property is perhaps the most important one, since we are using the loop invariant to show correctness. It also differs from the usual use of mathematical induction, in which the inductive step is used infinitely; here, we stop the "induction" when the loop terminates.

HeapSort loop invariant

$Build - Max - Heap(A)$

1 $heap\text{-}size[A] = length[A]$
2 **for** $i = \lfloor length[A]/2 \rfloor$ **downto** 1
3 $\quad$ MAX-HEAPIFY(A, i)

To show why BUILD-MAX-HEAP **works correctly, we use the following loop invariant:**

At the start of each iteration of the for loop of lines 2– 3, each node $i + 1, i + 2, \ldots, n$ is the root of a max-heap.

Loop Invariant Proof

At the start of each iteration of the for loop of lines 2– 3, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

We need to show that this invariant is true prior to the first loop iteration, that each iteration of the loop maintains the invariant, and that the invariant provides a useful property to show correctness when the loop terminates.

Initialization: **Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial max-heap.**

Maintenance: **To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.**

Termination: **At termination, $i = 0$. By the loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.**