CS6033 Lecture 2 Slides/Notes

Proof by Induction; Algorithm Design Example; Priority Queues (Implicit Binary Heaps & HeapSort) (Notes & Ch 6)

By Prof. Yi-Jen Chiang
CSE Dept., Tandon School of Engineering
New York University

1

* Prove by Induction

[Induction Basis] (IB): Statement is true for $n = 1$
Prove

[Induction Hypothesis] (IH): Assume: statement is true for $n = K$.

[Induction step] (IS): Prove: statement is true for $n = K+1$
using [IH] i.e. using the assumption that statement is true for $n = K$.

$S(1)$

---

1 2 3 - - -
$S(1)$ $S(2)$ $S(3)$ - -

IB: Prove: $\boxed{S(1)}$ is true.

IH $\to$ IS: Prove $\boxed{S(K) \Rightarrow S(K+1)}$ $\forall K \geq 1$

$\boxed{\text{Basic Form}}$

2

Other Forms / Variants of Pf by Induction

e.g. Fibonacci number:

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ $\forall n \geq 2$

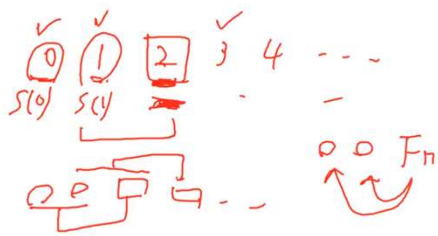
$$F_n = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

$\boxed{S(k-2) \wedge S(k-1) \Rightarrow S(k)}$

To prove any statement about F_n by Induction.

We need to prove 2 basis cases: $S(0)$: prove $S(0)$ is true

$S(1)$: " $S(1)$ is true.



Strongest Form:

$S(1) \cdots S(k)$ $\quad \underline{S(k+1)}$

$S(1) \wedge S(2) \wedge \cdots \wedge S(k) \Rightarrow S(k+1)$

3

Example: Proof by Induction with more than one base case.

Let g_n be defined as follows:

$$g_n = \begin{cases} 3 & \text{if } n=1 \\ 5 & \text{if } n=2 \\ 3g_{n-1} - 2g_{n-2} & \text{if } n \geq 3 \end{cases}$$

Prove that $g_n = 2^n + 1$ for all $n \geq 1$

Pf: (I) Induction Basis: ① when $n=1$.

$g_1 = 3 = 2^1 + 1$. ok.

② when $n=2$.

$g_2 = 5 = 2^2 + 1$. ok.

(II) Induction Hypothesis: Assume it's true

for $n = k-1$. i.e. $g_{k-1} = 2^{k-1} + 1$ —— (a) (where $k \geq 2$.)

and for $n = k$. i.e. $g_k = 2^k + 1$ —— (b)

(III) Induction Step: When $n = k+1$.

$$\begin{aligned} g_{k+1} &= 3g_k - 2g_{k-1} \\ &= 3(2^k+1) - 2(2^{k-1}+1) \\ &= 3 \cdot 2^k + 3 - 2^k - 2 \\ &= 2 \cdot 2^k + 1 = 2^{k+1} + 1 \end{aligned}$$

by (a) $g_{k-1} = 2^{k-1} + 1$

by (b) $g_k = 2^k + 1$

So the statement is true for $n = k+1$ ✓

$\Rightarrow g_n = 2^n + 1$ for all $n \geq 1$. ※

4

Some Algorithm Design Example:

Compute nth Fibonacci number.

i.e. Compute F_n.

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

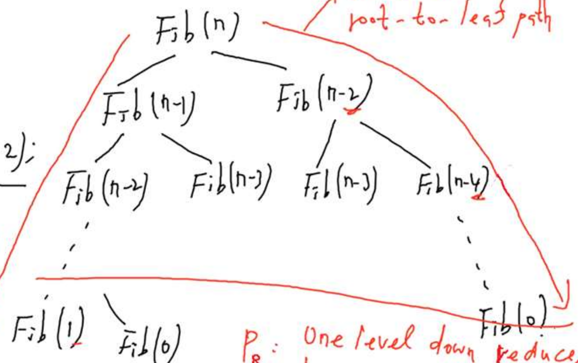
Alg 1: Fib(n)

```
{
  if (n == 0) return 0
  if (n == 1) return 1
  return Fib(n-1) + Fib(n-2);
}
```

call Fib(n).

Run-time Analysis

Recursion Tree:



P_R smallest root-to-leaf path

P_L: longest root-to-leaf path

complete binary tree.

P_R: One level down reduces n by 2, $n \to 0$. length: $\frac{n}{2}$.

5

Run time > size of the complete binary tree

$$= 1 + 2 + 2^2 + \cdots + 2^{\frac{n}{2}}$$

$$= \frac{2^{\frac{n}{2}+1} - 1}{2-1} = 2^{\frac{n}{2}+1} - 1 \geq 2^{\frac{n}{2}} = \Theta\left(2^{\frac{n}{2}}\right)$$

Exponential time !!

Improvement:

Observation: In Fib(n) recursion tree, many computations are repeated! We should avoid that.

Idea: Once we compute $F(K)$ then store it to avoid computing it again. (for all K)

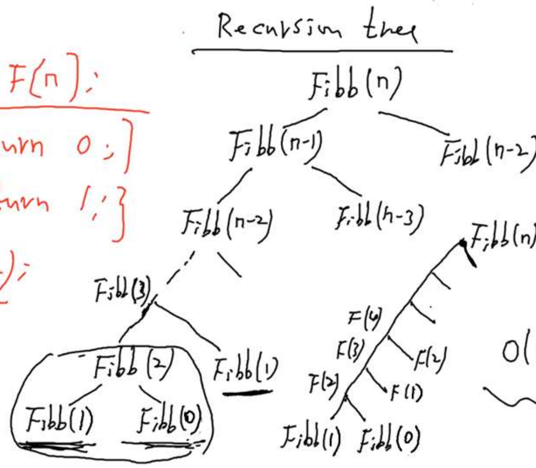
6

Alg 2 : 1. Create an array F[0 -- n], initialize each item to -1. (F[0] = F[1] = ... = F[n] = -1)

2. In function Fibb(k), check F[k] before computing it.

```
Fibb(n)
{ if (F[n] != -1) return F[n];
  if (n == 0) { F[0] ← 0. return 0; }
  if (n == 1) { F[1] ← 1. return 1; }
  F[n] ← Fibb(n-1) + Fibb(n-2);
  return F[n];
}
```

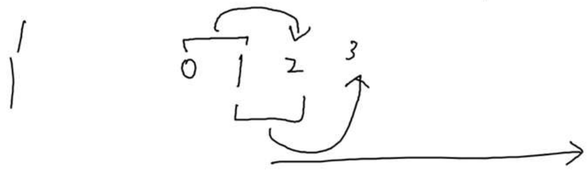
Recursion tree



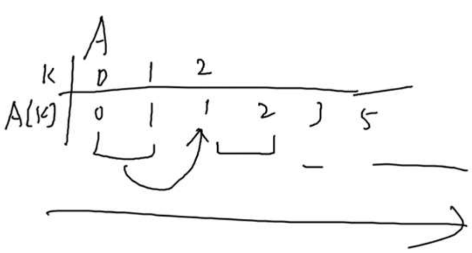
O(n) time!

7

Alg 3 : Iterative, bottom up.




```
A[0] ← 0
A[1] ← 1
For k = 2 to n.
  A[k] ← A[k-1] + A[k-2]
}
```



O(n) time

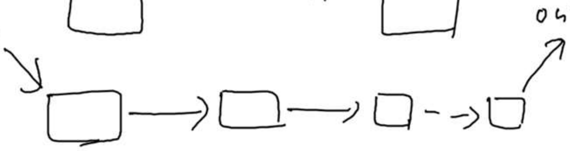
8

Quick Review of stack, queue, dequeue. array

Stack:

First-in -last-out

Queque,



First-in-First-out.

Doubly-Ended Queue:

Array,

9

Abstract Data Type (ADT)

Priority Queue (P. Q)

Max - P.Q Q.

1. Find_Max(Q)
2. Extract_Max(Q)
3. Insert(X, Q)
4. Increase_key(X, K, Q)
5. Delete(X, Q)

Min- P. Q Q

1. Find_Min(Q)
2. Extract_Min(Q)
3. Insert(X, Q)
4. Decrease_key(X, k, Q)
5. Delete(X, Q)

Example Data Structure for. P. Q

binary heap.

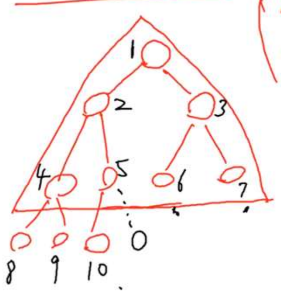
binomial heap
Fibonacci heap
⋮
(Advanced)

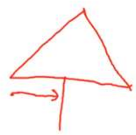
10

Implicit binary max-heap

(min-heap)

Using array to implement a binary tree (close to complete binary tree as much as possible)





A[1...n] to represent it.

root : A[1]

left_child(i) = 2i

right_child(i) = 2i+1

Parent(i) = $\lfloor \frac{i}{2} \rfloor$.

If A[ ] has n nodes what's the tree height?

height ≤ i+1

\# nodes in △ = $1+2+4+\dots+2^i = 2^{i+1}-1 \le n$

$i+1 \le \log_2(n+1) = O(\log n)$ ※

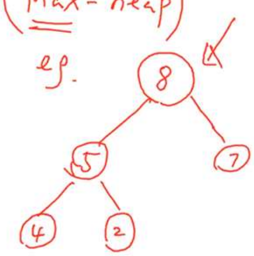
Maintain: counter n, for # of nodes in the heap.

11

Heap property : key(parent) ≥ key(child) (Partial order)

(Max-heap)

eg.

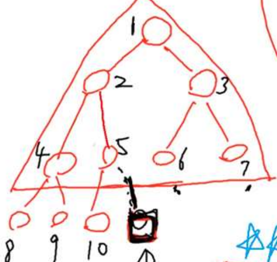


12

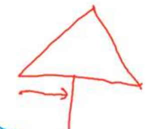
Implicit binary max-heap

(min-heap)

Using array to implement a binary tree (close to complete binary tree as much as possible)



insertion pt



$A[1 \ldots n]$ to represent it.

Key: Always insert/delete at the end of array !!

root : $A[1]$

left_child$(i) = 2i$

right_child$(i) = 2i+1$

parent$(i) = \lfloor \frac{i}{2} \rfloor$.

If $A[\;]$ has n nodes what's the tree height ?

i complete binary tree

height $\le i+1$

nodes in $\triangle = 1+2+4+\cdots+2^i = 2^{i+1}-1 \le n$.

$i+1 \le \log_2(n+1) = O(\log n)$ ※

13

Abstract Data Type (ADT)

Priority Queue (P.Q)

Max-P.Q Q.	Min-P.Q Q
1. Find_Max(Q) : $O(1)$	1. Find_Min(Q)
2. Extract_Max(Q)	2. Extract_Min(Q)
3. Insert(x, Q) $O(\log n)$	3. Insert(x, Q)
4. Increase_key(x, k, Q)	4. Decrease_key(x, k, Q)
5. Delete(x, Q) $O(\log n)$	5. Delete(x, Q)

Example Data Structure for P.Q

binary heap.

binomial heap

Fibonacci heap

⋮

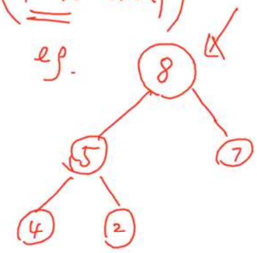
(Advanced)

14

Heap property : key(parent) $\geq$ key(child) (Partial order)

(Max-heap)

e.g.



Max-Heapify(i, A) :

Make the subtree rooted at A[i] to be a legal max heap. assuming that its left subtree & right subtree are each a legal max heap.

Compare A[l] and A[r]. take the larger one. call it A[large].

Compare A[i] & A[large]. → if (A[large] > A[i])
{ swap A[large] with A[i]
Max-Heapify(large, A)
}

$O(\log n)$ time

15

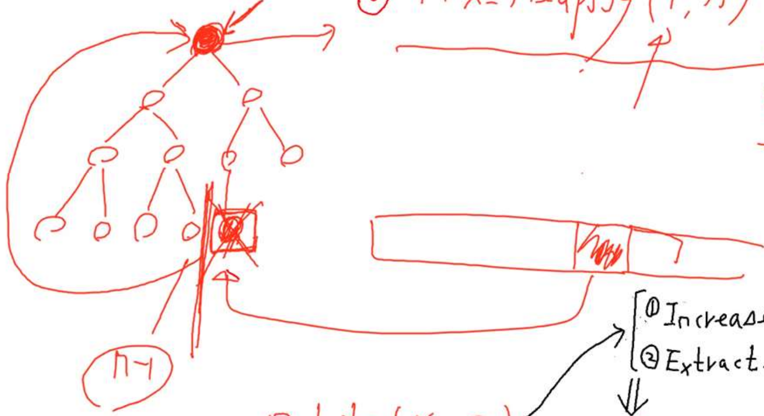
Extract_Max (Q):

① Take out A[1] (root).
② A[1] ← A[n] (Q is now A[1..n-1])
③ Max_Heapify (1, A)

Delete from the end of array

$O(\log n)$ time

$O(\log n)$ time



Delete(x, Q) → ① Increase_key(x, ∞, Q)
② Extract_Max (Q)

⇓

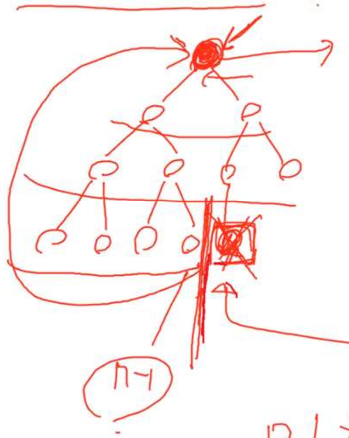
$O(\log n)$ time

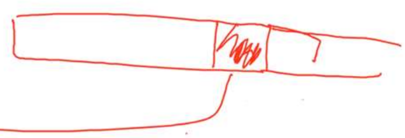
16

Extract_Max(Q):

① Take out $A[1]$ (root)
② $A[1] \leftarrow A[n]$
③ Max_Heapify(1, A)

$O(\log n)$ time





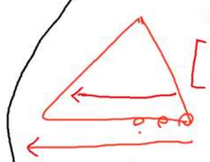
Delete(X, Q): $O(\log n)$ time

Heap Sort

M1: ① Perform n Insert( ) ops to insert n items to Q
② Perform n Extract_Max(Q) ops.

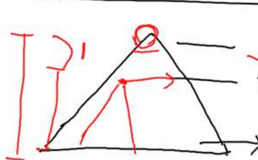
M2: In M1, replace ① by Build-Max-Heap( )

For $i \leftarrow n$ to 1 do
Max-Heapify(i, A)



17

Anylasis of Build-Max-Heap( ): $O(n)$ time.



1 node : heigh = $\log n$.
2 nodes : " = $(\log n - 1)$
⋮
$\le \frac{n}{2}$ nodes with height 1.

$\le \frac{n}{2}$ nodes with height 1
$\le \frac{n}{4}$ " " 2
⋮
$\le \frac{n}{2^i}$ nodes with height i.

Total time

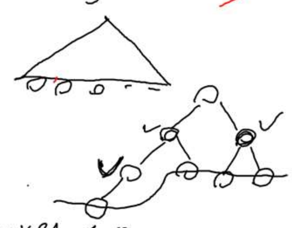
$$\le 1\cdot\frac{n}{2} + 2\cdot\frac{n}{4} + \cdots = \sum_{i\ge1}^{\log n} i\cdot\frac{n}{2^i} = S.$$

$$S = 1\cdot\frac{n}{2} + 2\cdot\frac{n}{4} + 3\,\frac{n}{8} + \cdots$$

$$-)\ \frac{1}{2}S = \qquad 1\cdot\frac{n}{4} + 2\cdot\frac{n}{8} + \cdots$$

$$\frac{1}{2}S = \frac{n}{2}\left(1+\frac{1}{2}+\frac{1}{4}+\cdots\right) \le \frac{n}{2}\cdot\frac{1}{1-1/2} = \frac{n}{2}\cdot 2 = n$$

$$\boxed{S = 2n = O(n)}$$



leaves $\le \frac{n}{2}+1$

18