Tirgul 6

Heaps
Induction

Heaps

- A binary heap is a nearly complete binary tree stored in an array object
- In a max heap, the value of each node ≥ that of its children
- (In a min heap, the value of each node ≤ that of its children)
- Since the height of a heap containing n elements is $\Theta(\log(n))$ the basic operations on a heap run in time that is proportional to the heaps height and thus take $O(\log(n))$

Heaps

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

Heaps

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


Mathematical Induction

- Mathematical induction is a method of mathematical proof typically used to establish that a given statement is true for all natural numbers, or for all members of an infinite sequence
- The simplest and most common form of mathematical induction proves that a statement holds for all natural numbers n and consists of two steps:
  1. The basis: showing that the statement holds when n = 0.
  2. The inductive step: showing that if the statement holds for n = k, it also holds for n = k + 1.

Mathematical Induction

- This method works by:
  1. First proving the statement is true for a starting value
  2. Then proving that the process used to go from one value to the next is valid.
- If these are both proven, then any value can be obtained by performing the process repeatedly
- For example, suppose we have a long row of dominos standing, and we can be sure that:
  1. The first domino will fall.
  2. Whenever a domino falls, its next neighbor will also fall.

➲ We can conclude that all dominos will fall.

Mathematical Induction

- Suppose we wish to prove the following statement:

$$S(n) = 0 + 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

- Proof by induction:
- Check if it is true for $n = 0$ $\quad S(0) = \frac{0 \cdot 1}{2} = 0$
- Assume the statement is true for $n = m$

$$S(m) = 0 + 1 + \ldots + m = \frac{m(m+1)}{2}$$

- Adding $m+1$ to both sides gives:

$$0 + 1 + \ldots + m + m + 1 = \frac{m(m+1)}{2} + m + 1$$

$$= \frac{m(m+1)}{2} + \frac{2(m+1)}{2} = \frac{(m+1)(m+2)}{2} = S(m+1)$$

Mathematical Induction

- Note that it has not been proven as true: we made the assumption that $S(m)$ is true, and from that assumption we derived $S(m+1)$. Symbolically, we have shown that:

$$S(m) \rightarrow S(m+1)$$

- However, by induction, we may now conclude that the statement $S(n)$ holds for all natural numbers n.

we have showed $S(0)$
we have showed that $S(0) \rightarrow S(1)$
we have showed that $S(1) \rightarrow S(2)$
etc.

Generalization

- A common generalization is proving that a claim is true for all $n \geq c$
  1. Showing that the statement holds when $n = c$
  2. Showing that if the statement holds for $n = m \geq c$, it also holds for $n = m+1$
- This can be used for showing that $n^2 \geq 10n$ for all $n \geq 10$
  1. For n==10, $n^2 = 10*10 = 10n$
  2. Assuming the statement holds for $n = m \geq 10$, we get:

$$(m+1)^2 = m^2 + 2m + 1 \geq 10m + 2m + 1 > 10(m+1)$$

$$(m \geq 10 \Rightarrow 2m + 1 > 10)$$

Complete Induction

- Another form of mathematical induction is the ***complete-induction*** (also called ***strong-induction***)
- In complete induction, the inductive hypothesis, instead of simply $H(n-1)$ is $\forall i \in \{1,...,n-1\} H(i)$
- (we have a stronger inductive hypothesis, hence the name ***strong-induction***)
- The complete induction steps are therefore:
  1. Showing that the statement holds for $n = 0$ (or $n = c$)
  2. Showing that if the statement holds for all $c \leq n \leq m$ then the same statement also holds for $n = m+1$

The game of Nin

- Rules:
  - Two players throw a number of stones on the ground
  - At each turn, a player removes one two or three stones
  - The one to remove the last stones loses
- Proposition:
  - The second player has a winning strategy **iff** the number of stones is $4k+1$, otherwise the first player has a winning strategy
- Proof:
  - Base case – there is only one stone, the second player wins $(1=4k+1)$
  - Induction: Assume that $P(n)$ is true for all $1 \leq n \leq m$ and prove that $P(n+1)$ is true as well

The game of Nin

- We have four possible cases:
  1. $n+1 = 4k+1$, We have already showed $P(1)$ to be true, so we assume that $n+1 \geq 5$. The first player can lift either one, two, or three stones, leaving either $4k, 4(k-1)+3, 4(k-1)+2$ respectively. By the induction hypothesis, the person who plays next has a winning strategy.
  2. $n+1 = 4k$, The first player can remove just three stones, leaving $n = 4(k-1)+1$. The strong induction hypothesis asserts that the second player loses.
  3. $n+1 = 4k+2$, The first player can remove just one stone, leaving $n = 4k + 1$. The strong induction hypothesis asserts that the second player loses.
  4. $n+1 = 4k+3$, The first player can remove two stones, leaving $n = 4k + 1$. The strong induction hypothesis asserts that the second player loses.

Loop Invariants

- A loop invariant is statement that is true when a program enters a loop, remains true in each iteration of the body of the loop, and is still true when control exits the loop.
- Understanding loop invariants can help us analyze programs, check for errors, and derive programs from specifications.
- The loop invariant technique is a derived from the mathematical induction:
  1. We first check that the statement is true when the loop is first entered
  2. We then verify that if the invariant is true after n times around the loop, it also true after n+1 times

Loop Invariants

```
public static int factorial(int num) {
    int P = 1;
    int C = 0;
    while (C < num) {       // P = C! (our loop invariant)
        C++;
        P *= C;
    }                       // P = C! (our loop invariant)
    return (P);
}
```

- Our loop invariant is true when we first enter and last leave the loop

Insertion sort

- The outer loop of insertion sort is:
  `for (outer = 1; outer < a.length; outer++) {…}`
- The invariant is that all the elements to the left of outer are sorted *with respect to one another*
  - For all i < outer, j < outer, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase outer, a[outer-1] becomes to its left; we must keep the invariant true by inserting a[outer-1] into its proper place
  - This means:
    1. Finding the element's proper place
    2. Making room for the inserted element (by shifting over other elements)
    3. Inserting the element

Code for Insertion Sort

```
public static void insertionSort(int[] array)
    int inner, outer;
for(outer=1; outer<array.length; outer++) {
    int temp = array[outer];
    inner = outer;
    while(inner>0 && array[inner-1] >= temp) {
         array[inner] = array[inner - 1];
         inner--;
    }
         array[inner] = temp;
    // Invariant: For all i < outer, j < outer, if i < j then a[i] ≤ a[j]
}
}
```


Insertion Sort - Loop Invariants

- The loop invariant is true before the loop is first executed
- If the loop invariant holds after n times around the loop, the inner loop makes sure we insert the n+1 element in place, therefore the loop invariant also true after n+1 times