CSC265 Notes

Max Xu

'25 Fall

Contents

§1 Day 1: Review and ADTs (Sept 3, 2025)

§1.1 ADTs

Abstract data types (ADTs) are a mathematical object, which has a set of operations. Examples include a set, a sequence, or a graph.

A *data structure* that implements an ADT provides a representation for the object in memory and algorithms for each operation. There may exist many different data structures that implement the same ADT, with varying running time for each operation.

Example 1.1 (Dictionary ADT)

The ADT consists of:

- **Object**: a set of elements, with each having a unique key from totally ordered universe U
- **Operations**:

  Insert(S, x) adds an element with key x to the set S if S does not contain an element with key x

  Delete(S, x) removes an element with key x from S if it exists

  Search(S, x) returns a pointer to the element in S with key x, or nil if such an element doesn't exist

Here we are not worried about key-value pairs, although a specific implementation of this ADT might. We could implement this as a set of keys, with no associated element for each key.

An example implementation of dictionary ADT using singly linked list (unsorted) can have the following properties:

§1.2 Review

Let $t(x)$ be the number of steps taken by an algorithm A on input x.
Let the worst case step complexity of A be $T(n)$, where

$$T(n) = \max\{t(x) \mid x \text{ is an input of size } n\}$$

Typically, this is very hard to determine exactly, which is why asymptotic notation is used instead, as it still captures how quickly $T(n)$ grows with respect to n.

$T(n) \in O(f(n))$ if there exists a constant $c, n_0 \in \mathbb{N}$, for all $n \in \mathbb{N}$ such that when $n > n_0$, $T(n) \leq cf(n)$. $T(n) \in \Omega(f(n))$ if there exists a constant $c, n_0 \in \mathbb{N}$, for all $n \in \mathbb{N}$ such that when $n > n_0$, $cf(n) \leq T(n)$. $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$.

For upper bound, show that there exists positive constant c and for all n large enough, for every input of size n, the algorithm takes at most $cf(n)$ steps. We write $T(n) \in O(n)$.

For lower bound, show that there exists positive constant c and for all n large enough, there exists some input of size n that makes the algorithm take at least $cf(n)$ steps. We write $T(n) \in \Omega(n)$.

BubbleSort($A[1...n]$**)**

```
1: last = n
2: sorted = False
3: while not sorted do
4:     sorted = True
5:     for j = 1 to last − 1 do
6:         if A[j] > A[j + 1] then
7:             swap A[j] and A[j + 1]
8:             sorted = False
9:         end if
10:    end for
11:    last = last − 1
12: end while
```

Upper Bound

The outer loop can occur at most n times, as last starts at n and decrements by 1 every time the outer loop runs. The inner loop goes from 1 to last $-$ 1, meaning there are at most last $-$ 1 iterations of the inner loop. This can be written as the sum

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

meaning for this algorithm $T(n) \in O(n^2)$.

Lower Bound

We can pick the list $A = [n, n-1, \ldots, 2, 1]$. This list has the property that the first element is the largest number, barring the sorted block of size $i-1$ present at the end of the array at the start of the i-th iteration.[1] This means that it will take $n-i$ swaps, which means it takes

$$\sum_{i=1}^{n}(n-i) = n^2 - \frac{n(n+1)}{2} = \frac{n^2-n}{2}$$

steps total, meaning for this algorithm $T(n) \in \Omega(n^2)$.

Theta Bound

As $T(n) \in \Omega(n^2)$ and $T(n) \in O(n^2)$, we have $T(n) \in \Theta(n^2)$.

[1] i think iterations start at 1 in this course?

§2 Day 2: Priority Queues (Sept 5, 2025)

Last class we were looking at worst case analysis for the dictionary ADT.

Sorted Array

Insert(A, x) $\Theta(n)$, necessary to shift elements 'greater' than x right by 1

Delete $\Theta(n)$, necessary to shift elements 'greater' than x left by 1

Search(A,x) Use binary search, known to take $\Theta(\log n)$

Direct Access Table

Suppose universe of keys $U = \{1, \ldots, n\}$ and n is small. As array indexing takes $\Theta(1)$ time, all operations with the ADT take $\Theta(1)$ time.

We have a slot for every single key in the universe, meaning we could hold the keys themselves, pointers to objects in the set, or even numbers to extend to a multiset, etc.

§2.1 Priority Queue ADT

The object is a set of elements each with a key (sometimes called priority, for consistency with CLRS), with the following operations

Insert(S, x) adds an element with key x to S

Max(S) returns the largest key

Remove(S) removes element with largest key and returns key

§2.2 Binary Heap ADT

The set S is stored in a complete binary tree (every level is filled except possibly for the lowest level, which is filled from left to right)

A complete binary tree with height h has $2^h \leq n \leq 2^{h+1}$ nodes. With n nodes, it has height $\lfloor \log n \rfloor$. A max heap is a complete binary tree that satisfies the **max heap** property, where the key of each node is greater than or equal to the keys of its children (if they exist).

§2.3 Array Representation of Binary Heaps

A binary heap can be represented as an array of length equal to the number of nodes, $A.heapsize$. In this course arrays are indexed starting at 1. Advantages include no need to store papers

Theorem 2.1. Given an index i corresponding to a node in a binary heap represented as an array A, we have $left(i) = 2i, right(i) = 2i + 1, parent(i) = \lfloor \frac{i}{2} \rfloor$

The max heap property for this representation is as follows:

$$\text{For all integers } i,\ 1 < i \leq A.heapsize, A[\lfloor \tfrac{i}{2} \rfloor] \geq A[i]$$

§2.3.1 Operations

Max(A) return A[1], $\Theta(1)$ worst case

Insert(A, x) Increment $A.heapsize$, set $A[A.heapsize] = x$. While the new inserted node has key larger than parent key, swap places with parent.[2]

IncreaseKey(A, i, k) Increase $A[i]$ to k, takes $\Theta(\log A.heapsize)$, used for the 'bubbling up' procedure in INSERT. [3]

MaxHeapify(A, i)

Preconditions: $A[i].left$ and $A[i].right$ are max heaps$A[i]$ may be smaller exactly one of its children.

Postconditions: $A[i]$ is the root of max-heap with same elements

Swap the root with the larger of the children, so that the violation moves down the tree. Similar loop invariant to earlier, worst case runtime $\Theta(\log n)$.

ExtractMax(A) Set $r = A[1]$, for returning at the end. Swap $A[A.heap.size]$ (last element) and $A[1]$. Then decrement A.heap.size, and call MaxHeapify(A, 1), as putting the last element in the first index may violate the max heap property. Finally return the r.

§2.3.2 Building a Max Heap

Problem 2.2. Given an array $A[1..n]$ of n keys in arbitrary order, we want to modify A-in place such that it is a MaxHeap.

The naïve non-inplace solution is to perform n $Insert(A, x)$ operations, which will take $\Theta(n \log n)$ steps. We could also reverse sort the array in decreasing order, which ensures the max-heap property is satisfied since the element at each index, which generally takes $O(n \log n)$.

[2] To prove correctness, the loop invariant is that below level n, there are no violations of the max heap property, then you move n to 0.

[3] I don't really understand what k is here, will ask soon