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, Array Representation (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, which we can 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 $2^h \le n \le 2^{h+1}$ 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:

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

§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

§3 Day 3: Assignment Expectations, Heaps (Sept 8, 2025)

The CLRS chapter on binomial heaps is posted on Quercus, because it was removed after the 2nd chapter.

§3.1 Assignment Expectations

For reference, the solutions to Assignment 1 takes up roughly 4-5 pages. TLDR: Emulate the style of proof found in CLRS

Bounds

- Need to show upper and lower bound **separately**[4]

- To show $f(n) \in O(n)$ or $\Omega(n)$, we don't need to find specific values for c, n_0, just need to simplify f such that it's clear

Algorithms

To give/describe an algorithm:

- Start with higher-level details, explain its phases

- Describe all parts of the algorithm in *English*, which can easily be converted to pseudo-code. Don't miss edge cases, and pointer updates when the happen. Your peers should be able to understand it clearly.

- Producing pseudo-code is **optional**, unless explicitly asked for

- You can use algorithms you previously encountered in class or in past courses

- Diagrams are helpful, but cannot replace your English description

Proof of Correctness

- Always say what you are proving. For long results, introduce lemmas, for short results, that is unnecessary. No flow of consciousness!

- Make clear what proof technique you're using: "We will prove by induction on the length n . . . ", "Suppose for contradiction that . . . "

- We can be less formal, we don't need to explicitly state what P is, no CSC240-indentation is strictly required. Still use indentations to break up paragraphs though, for clarity.

---

[4]note that no technique exists to directly show Θ, without showing O and Ω beforehand

§3.2 Building Heaps

We want an $\Theta(n)$ algorithm to make an array A possess the MaxHeap property (in-place).

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

1: A.heapsize = n
2: **for** $i = \lfloor \frac{n}{2} \rfloor$ to 1 **do**
3:      MaxHeapify($A[i]$)
4: **end for**

The loop invariant is that at the start of the iteration of the floor loop, $A[i+1], \ldots, A[n]$ is the root of a MaxHeap.

At most n calls will be made to the MAXHEAPIFY function, each taking $O(\log n)$ steps in the worst case, meaning the total steps is $O(n \log n)$. This analysis isn't wrong per-se, it is a valid upper bound, just not a tight one.

The reason the bound wasn't tight is because for the lower levels, we are performing a lot less swaps than $\log n$. We observe that there are at most 2^{h-i} nodes at the i-th level (note that the root is considered to be at level 0). MAXHEAPIFY on a node of height i takes at most ci steps.

$$
\begin{aligned}
T(n) &= c \cdot \sum_{i=1}^{h} 2^{h-i} i \\
&= c \cdot 2^h \sum_{i=1}^{h} \frac{i}{2^i} \\
&\leq cn \sum_{i=1}^{\infty} \frac{i}{2^i} \\
&= 2cn \in O(n)
\end{aligned}
$$

For the lower bound, at least $\lfloor \frac{n}{2} \rfloor$ calls to MAXHEAPIFY are made, because of the loop terminatwith each taking at least 1 step. Hence it is $\Omega(n)$.

## §3.3 Binomial Heaps

We want *mergeable* priority queues, which support an additional operation.

Union(S_1, S_2**)** Given two priority queues S_1 and S_2, merge them into a single queue with the union of their elements. Assume the elements in S_1 and S_2 are disjoint.

Definition 3.1 (Binomial Trees). A binomial tree of degree k is defined recursively:

- B_0 is a single node

- B_k consists of two binomial trees B_{k-1}, with the root of 1 being the leftmost child of the root of the other
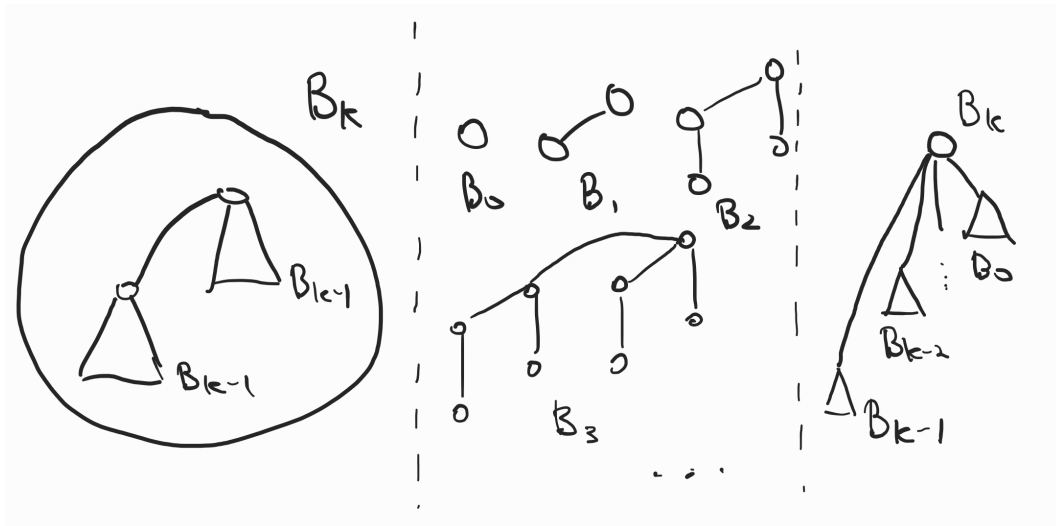
For a binomial tree of degree k,

1. There are 2^k nodes

2. The height is k

3. There are exactly $\binom{k}{i}$ nodes at depth i[5]

4. The root has degree k and its i-th child is the root of a binomial tree B_i (children are labelled $k - 1$ to 0 left to right)

Defining the operations this way makes the operations quite simple, we didn't have time to go over them this class.

---

[5]follows from combinatorics facts

§4 Day 4: Binomial Heaps (Sept 10, 2025)

Last week, we looked at binary heaps. Today we will look at binomial heaps, whose definition were given last class. To store n nodes where n were not a power of 2, write n in binary

$$n = b_k 2^k + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

with $b_i \in \{0, 1\}$. The collection of binomial trees (binomial forest) where there is a binomial tree B_i of degree i for each b_i when $b_i = 1$ would have n nodes.

A **binomial heap** is a binomial forest, where the roots of the binomial trees are in a LinkedList sorted by strictly increasing degree. Each binomial tree satisfies the max-heap property.

To represent a binomial heap in memory, have each node x store its key, degree, parent, left child (pointer), and right sibling (pointer to the sibling immediately to its right). Note that for the roots of trees B_i, there will be no right-sibling, so we can use the right-sibling field to achieve LinkedList behavior.

Link(H_1, H_2) : link together two binomial trees with the same degree. need to update the degree $(+1)$ and left child of the tree with the larger root key to the tree with the smaller root key. update the right sibling and parent of the tree with smaller root key. $\Theta(1)$, as this is a constant number of pointer updates.

Union(H_1, H_2) : merge LinkedList of H_1 and H_2 sorted by increasing degree. link trees together by increasing degree until at most 1 tree of each degree link trees together by increasing degree until at most 1 tree of each degree link trees together by increasing degree until at most 1 tree of each degree. H_1 has t_1 trees, H_2 has t_2 trees, merging would take $O(t_1 + t_2)$, linking would require $O(t_1 + t_2)$, the sum is $O(\log n)$

Insert(H, x) : Create a single node N with key x, then call UNION(H, N). Note that if $n = 2^{k-1}$, then it would take $O(\log n)$, since you will be performing at least $\log n$ merges. However if $n = 2^k$, then the new node will just get added to the linkedlist in constant time. For this reason you can't get a tight Θ bound on this operation. Also need to update the max pointer here, if the new inserted node is greater than MAX pointer.

Max(H) traverse the LinkedList and return the largest, takes $O(\log n)$. we can make this $\Theta(1)$, by maintianing a pointer to the root with the largest key.

ExtractMax(H) : Traverse the roots to find binomial tree B_i with the largest key. Remove the root of B_i, create binomial heap H from its children. To ensure it's valid, reverse the order, this means sorted by increasing degree, then call UNION(H, H'). Traverse roots to find the largest key, to update the MAX pointer.

Next week, we will look at balanced binary search trees.