Queens' College Cambridge

Algorithms

Alistair O'Brien

Department of Computer Science

May 25, 2020

Contents

1 Growth Of Functions

1.1 Asymptotic notation

1.1.1 O-notation

The O-notation provides an asymptotic upper bound.

Definition 1.1.1. For a given function $g(n)$, we denote $O(g(n))$ as the set of functions

$$O(g(n)) = \{f(n) : \exists N, b > 0. \forall n > N. 0 \leq f(n) \leq bg(n)\}.$$

DIAGRAM HERE

- O-notation is not tight, for example $n \in O(n^2)$.

Macro convention

- A set in a formula represents an anonymous function in the set.
- For example

$$f(n) = n^3 + O(n^2) \iff \exists g(n) \in O(n^2). f(n) = n^3 + g(n).$$

and

$$n^2 + O(n) = O(n^2) \iff \forall f(n) \in O(n). \exists g(n) \in O(n^2). n^2 + f(n) = g(n).$$

1.1.2 Ω-notation

The Ω-notation provides an asymptotic lower bound.

Definition 1.1.2. For a given function $g(n)$, we denote $\Omega(g(n))$ as the set of functions

$$O(g(n)) = \{f(n) : \exists N, a > 0. \forall n > N. 0 \leq ag(n) \leq f(n)\}.$$

DIAGRAM HERE

- Ω-notation is not tight, for example $n^2 \in \Omega(n)$.

1.1.3 Θ-notation

The Θ-notation provides an asymptotic tight bound.

Definition 1.1.3. For a given function $g(n)$, we denote $\Theta(g(n))$ as the set of functions

$$\Theta(g(n)) = \{f(n) : \exists N, a, b > 0. \forall n > N. 0 \leq ag(n) \leq f(n) \leq bg(n)\}.$$

- Note that

  $$\Theta(g(n)) = \{f(n) : f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n))\}.$$

  Hence $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$. So $\Theta(g(n)) \subset O(g(n))$ and $\Theta(g(n)) \subset \Omega(g(n))$

Theorem 1.1.1. For all functions $f(n)$, $g(n)$, we have

$$f(n) \in \Theta(g(n)) \iff f(n) \in \Omega(g(n)) \wedge f(n) \in O(g(n)).$$

1.1.4 o-notation and ω-notation

The o and ω notations are used to denote an upper (and lower bound) that is not asymptotically tight.

Definition 1.1.4. For a given function $g(n)$, we denote $o(g(n))$ as the set of functions

$$o(g(n)) = \{f(n) : \exists N, b > 0. \forall n > N. 0 \leq f(n) < bg(n)\}.$$

Definition 1.1.5. For a given function $g(n)$, we denote $\omega(g(n))$ as the set of functions

$$\omega(g(n)) = \{f(n) : \exists N, a > 0. \forall n > N. 0 \leq ag(n) < f(n)\}.$$

1.1.5 Properties

Transitivity

For all functions $f(n), g(n), h(n)$, we have

$$\begin{aligned}
f(n) \in \Theta(g(n)) \wedge g(n) \in \Theta(h(n)) &\implies f(n) \in \Theta(h(n)) \\
f(n) \in O(g(n)) \wedge g(n) \in O(h(n)) &\implies f(n) \in O(h(n)) \\
f(n) \in \Omega(g(n)) \wedge g(n) \in \Omega(h(n)) &\implies f(n) \in \Omega(h(n))
\end{aligned}$$

Reflexivity

For all functions $f(n)$

$$f(n) \in \Theta(f(n))$$
$$f(n) \in O(f(n))$$
$$f(n) \in \Omega(f(n))$$

Symmetry

For all functions $f(n), g(n)$

$$f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n)).$$

Transpose symmetry

For all functions $f(n), g(n)$

$$f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$$

1.2 Solving recurrence relations

1.2.1 The substitution method

- The substitution method comprises of the following steps
  1. Guess the form of the solution
  2. Verify by induction
  3. Solve for constants

Example 1.2.1. Solve

$$T(n) = 2T\left(\lfloor n/2 \rfloor\right) + n.$$

We "guess" the solution is $T(n) \in O(n \log_2 n)$, that is to say

$$T(n) \leq bn \log_2 n.$$

for all $n > N$. We verify this by strong induction. For simplicity, we let $N = 1$ and define

$$T(1) = 1.$$

For our inductive proof, by the definition of big-O, we consider $n > 1$.

Proof.
Base Case. When $n = 2$, we have

$$T(2) = 4 \leq 2b,$$

for some sufficiently large $b \geq 2$. For our other base case, we have $n = 3$, yielding

$$T(3) = 5 \leq 3b,$$

for some sufficiently large $b \geq 2$.
Inductive Step. We wish to show that $[\forall k \in \{2, \ldots, n\} . P(k)] \implies P(n+1)$. Let us assume that $P(k)$ holds for all $k \in \{2, \ldots, n\}$, that is to say

$$T(k) \leq bk \log_2 k.$$

We wish to show that $P(n+1)$ holds. Let us note that $\lfloor n + 1/2 \rfloor \in \{2, \ldots, n\}$. Instantiating for $k = \lfloor n + 1/2 \rfloor$ gives us

$$T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) = b \left\lfloor \frac{n+1}{2} \right\rfloor \log_2 \left\lfloor \frac{n+1}{2} \right\rfloor.$$

So we have

$$\begin{aligned}
T(n+1) &= 2T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + (n+1) \\
&= 2\left[b \left\lfloor \frac{n+1}{2} \right\rfloor \log_2 \left\lfloor \frac{n+1}{2} \right\rfloor\right] + (n+1) \\
&= b(n+1) \log_2 \frac{n+1}{2} + (n+1) \\
&= b(n+1) \log_2 (n+1) - b(n+1) + (n+1) \\
&\leq b(n+1) \log_2 (n+1)
\end{aligned}$$

Hence $P(n+1)$ holds.
CONCLUSION

□

From our inductive proof, we note that $b \geq 2$. For simplicity, let $b = 2$ and $N = 1$. We have now shown that $T(n) \in O(n \log_2 n)$.

1.2.2 The recursion tree method

EXAMPLE

1.2.3 The master theorem

Theorem 1.2.1. (**Master theorem**) Let $a, b \geq 1$ be some arbitrary constants. Let $f(n)$ be some asymptotically positive function, and let $T(n)$ be defined by the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Then either

1. **Case 1**:
$$f(n) \in O(n^{\log_b a - \varepsilon}) \implies T(n) \in \Theta(n^{\log_b a}),$$
for some $\varepsilon > 0$.

2. **Case 2**:
$$f(n) \in \Theta(n^{\log_b a}) \implies T(n) = \Theta(n^{\log_b a} \log_2 n).$$

3. **Case 3**:
$$f(n) \in \Omega(n^{\log_b a + \varepsilon}) \wedge af\left(\frac{n}{b}\right) \leq cf(n) \implies T(n) \in \Theta(f(n)),$$
for some $\varepsilon > 0$.

Intuitively, we're simply comparing $f(n)$ with $n^{\log_b a}$ (the number of leaves in the recursion tree). In case 1, if $f(n)$ is polynomially smaller (by a factor of n^{ε}) than $n^{\log_b a}$, then $T(n) \in \Theta(n^{\log_b a})$. In case 2, the two functions are similar, we multiply by a logarithmic factor. In case 3, if $f(n)$ is larger, then $T(n) \in \Theta(f(n))$.

2 Divide-and-Conquer

2.1 Divide-and-conquer

In a divide-and-conquer approach, we recursively solve the problem using the following steps:

1. **Divide** the problem into one or more subproblems (of smaller size).
2. **Conquer** subproblems by solving them recursively. If the subproblem sizes are small enough, we solve the subproblem trivially (a base case).
3. **Combine** the solutions to the subproblems into the solution for the original problem.

2.2 Divide-and-conquer algorithms

2.2.1 Binary Search

- **Problem**: Given the sorted array A of n elements and an element x, find the index of x in A.
- Divide and conquer approach:
  - **Divide**: Compare x with the middle element of A, denoted by $A[\lfloor \ell/2 \rfloor]$.
  - **Conquer**: If $x < A[\lfloor \ell/2 \rfloor]$, recurse into the left subarray, if $x > A[\lfloor \ell/2 \rfloor]$ recurse into the right subarray.
  - **Combine**: No work is needed to combine, simply return the result from the recursive call.

Algorithm 1 Binary Search

```
function BINARY-SEARCH(A, x, l, r)
    if l < r then
        ▷ ASSERT: The subarray A[l : r + 1] is not empty.
        m ← ⌊((r+1)−l)/2⌋
        if x < A[m] then
            ▷ ASSERT: x is in the subarray A[l : m].
            return BINARY-SEARCH(A, x, l, m − 1)
        else
            if x > A[m] then
                ▷ ASSERT: x is in the subarray A[m + 1 : r + 1].
                return BINARY-SEARCH(A, x, m + 1, r)
            else
                ▷ ASSERT: x = A[m].
                return m
            end if
        end if
    end if
    ▷ ASSERT: The subarray A[l : r + 1] is empty.
    return -1
end function
```

- Analysis: By inspection, we have the recurrence relation

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

We note that in the case of the Master theorem, we have $a = 1$, $b = 2$ and $f(n) = \Theta(1)$. Let us note that

$$n^{log_2 1} = n^0 = 1.$$

Hence

$$f(n) \in \Theta(n^{\log_2 1}) = \Theta(1),$$

holds, so by **case 2** of the Master theorem, we have

$$T(n) = \Theta(\log_2 n).$$

2.2.2 Exponentiation

- **Problem**: Given the real number x and the integer $n \geq 0$, compute x^n.

- Naïve approach:

```
let rec pow x n = if n = 0 then 1
                  else x *. pow x (n - 1)
```

By inspection, we perform n operations, yielding a time complexity of $\Theta(n)$.

- Divide and conquer approach:

  - **Divide**: We note that

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{if n is even} \\ x \cdot x^{n-1/2} \cdot x^{n-1/2} & \text{otherwise} \end{cases}.$$

    So determine whether n is even and consider the subproblem $x^{n/2}$ or $x^{n-1/2}$.

  - **Conquer**: Recursively calculate $x^{n/2}$ (or $x^{n-1/2}$).

- **Combine**: Combine according to the cases noted in **divide** and return.

- ```
  let rec pow x n = match n with
      | 0 -> 1
      | n when even n -> power (x *. x) (n / 2)
      | n -> x *. power (x *. x) (n / 2)
  ```

- Analysis: By inspection, we have the recurrence relation

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

Similarly, to binary search, we have

$$T(n) = \Theta(\log_2 n).$$

2.2.3 Maximum-Subarray

- **Problem**: Given an array A of n real numbers, find the indices i, j where $1 \leq i \leq j \leq n$ such that

$$\sum_{k=i}^{j} A[k],$$

is maximised. That is the contiguous subarray with the largest sum.

- Naive approach: Consider all $\binom{n}{2}$ pairs of indices. Assuming the sum of a subarray is computed in $\Omega(1)$ time, then the approach takes $\Omega(n^2)$ time.

- Divide and conquer approach:
  - **Divide**: the subarray $A[l : r+1]$ into two subarrays $A[l : m]$ and $A[m+1 : r+1]$. The maximum subarray $A[i : j+1]$ then lies in either
    * The left subarray $A[l : m]$ such that $l \leq i \leq j < m$.
    * The right subarray $A[m : r+1]$ such that $m \leq i \leq j \leq r$.
    * Crossing the midpoint, such that $l \leq i < m < j \leq r$
  ```

  - **Conquer**: Recursively find the maximum subarrays of $A[l : m]$ and $A[m : r + 1]$. Use a non-recursive procedure to find the maximum subarray that crosses the midpoint.
  - **Combine**: Compare all three subarrays and return the largest.

Algorithm 2 Maximum Subarray

```
function Maximum-Subarray(A, l, r)
    if l ≠ r then
        m ← ⌊(r + 1) − l/2⌋
        (l_l, r_l, s_l) ← Maximum-Subarray(A, l, m − 1)
        (l_r, r_r, s_r) ← Maximum-Subarray(A, m, r)
        (l_c, r_c, s_c) ← Max-Crossing-Subarray(A, l, m, r)
        S ← {(l_l, r_l, s_l), (l_r, r_r, s_r), (l_c, r_c, s_c)}
        return argmax_{(l′,r′,s′)∈S} s′
    end if
    return (l, r, A[l])
end function
```

- Analysis: Let us denote the running time of Maximum-Subarray applied to an array A of length n by $T(n)$. By inspection, we have

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \underbrace{\Theta(n)}_{\text{Max-Crossing-Subarray}} & \text{otherwise} \end{cases}$$

  This has the solution $T(n) = \Theta(n \log_2 n)$ by the Master theorem.

3 Sorting

3.1 Insertion Sort

Algorithm 3 Insertion Sort

```
procedure Insertion-Sort(A)
    for i ← 1 to A.length − 1 do
        ▷ ASSERT: The subarray A[0 : i] is sorted
        k ← A[i]
        j ← i − 1
        while j > 0 ∧ A[j] > k do
            ▷ ASSERT: The subarray A[j + 1 : i + 1] ≥ k
            A[j + 1] ← A[j]
            j ← j − 1
        end while
        A[j + 1] ← k
    end for
end procedure
```

- **Correctness**: Prove the following loop invariant using induction on i:

$$P(i) = \text{The subarray } A[0:i] \text{ is sorted at the } i\text{th iteration..}$$

- **Analysis**: Complexity of $O(n^2)$.
- Other variants such as binary insertion sort, requires $O(n \log_2 n)$ comparisons, but $O(n^2)$ swaps.

3.2 Selection Sort

Algorithm 4 Selection Sort

```
procedure SELECTION-SORT(A)
    for i ← 0 to A.length − 1 do
        ▷ ASSERT: The sorted subarray A[0 : i] is the minimum subarray
        j ← i
        for k ← i + 1 to A.length − 1 do
            if A[k] < A[j] then
                j ← k
            end if
        end for
        ▷ ASSERT: A[j] is minimum element in subarray A[i :]
        swap A[i], A[j]
    end for
end procedure
```

- Number of comparisons is constant (regardless of input state).
- Performs the minimum number of swaps ($n - 1$).
- **Analysis**: Complexity of $\Theta(n^2)$.

3.3 Bubble Sort

Algorithm 5 Bubble Sort

```
procedure BUBBLE-SORT(A)
    complete ← false
    while not complete do
        complete ← true
        for i ← 0 to A.length − 2 do
            if A[i] > A[i + 1] then
                swap A[i], A[i + 1]
                complete ← false
            end if
        end for
    end while
end procedure
```

- Iterates through the array, swapping elements until no more swaps are required.
- **Analysis**: Complexity of $O(n^2)$ (but if sorted, has a running time of $O(n)$)

3.4 Merge Sort

- Divide and conquer approach:
  - **Divide**: the n-element array A into two subarrays $A[: m]$ and $A[m :]$ of $n/2$ elements.
  - **Conquer**: Recursively sort the two subarrays using merge sort.
  - **Combine**: Merge the two sorted subarrays to produce the sorted array.

Algorithm 6 Merge Sort

procedure MERGE(A, l, m, r)
    $n_l \leftarrow (m + 1) - l, n_r \leftarrow r - m$
    $A_l \leftarrow A[l : m + 1], A_r \leftarrow A[m + 1 : r + 1]$
    $i \leftarrow 0, j \leftarrow 0, k \leftarrow l$
    while $i < n_l \wedge j < n_r$ **do**
        $A[k++] \leftarrow A_l[i++]$ **if** $A_l[i] < A_r[j]$ **else** $A_r[j++]$
    end while
    while $i < n_l$ **do**
        $A[k++] \leftarrow A_l[i++]$
    end while
    while $j < n_r$ **do**
        $A[k++] \leftarrow A_r[j++]$
    end while
end procedure
procedure MERGE-SORT(A, l, r)
    if $l < r$ **then**
        $m \leftarrow \lfloor (l + r)/2 \rfloor$
        MERGE-SORT(A, l, m)
        MERGE-SORT$(A, m + 1, r)$
        MERGE(A, l, m, r)
    end if
end procedure

- **Correctness of Merge**: Prove the following loop invariant by induction on k:

$$\begin{aligned} P(k) = &\text{ The subarray } A[l : k] \text{ is the minimum sorted subarray and} \\ &A_l[i] \text{ and } A_r[j] \text{ are the smallest elements not in } A[l : k] \end{aligned}$$

- **Analysis of Merge**: Complexity of $\Theta(n)$. At worst $r - l$ comparisons and $n = (r + 1) - l$ assignments.

- **Analysis of Merge-Sort**: Recurrence relation of

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases},$$

  where $n = r+1-l$. Applying Master theorem yields a time complexity

$$T(n) \in \Theta(n \log n),$$

  and a space complexity of $\Theta(n)$.

- This is a top-down variant, bottom-up merge sort is INSERT HERE

3.5 Quicksort

- Divide and conquer approach:
  - **Divide**: Partition the array $A[l : r+1]$ into two subarrays $A[l : p]$ and $A[p + 1 : r + 1]$ such that $\forall x \in A[l : p].x \leq A[p]$ and $\forall x \in A[p + 1 : r + 1].A[p] < x$
  - **Conquer**: Recursively sort the two subarrays using quicksort.
  - **Combine**: Since the subarrays are sorted and from the partition property, no work is needed to combine them.

Algorithm 7 Quicksort

```
function Partition(A, l, r)
    x ← A[l]
    i ← l + 1
    for j ← l + 1 to r do
        if A[j] ≤ x then
            swap A[i], A[j]
            i++
        end if
    end for
    swap A[--i], A[l]
    return i
end function
procedure Quicksort(A, l, r)
    if l < r then
        p ← Partition(A, l, r)
        Quicksort(A, l, p)
        Quicksort(A, p + 1, r)
    end if
end procedure
```

- **Correctness of Partition**: Prove the following loop invariant using induction on j:

$$P(j) = \forall k \in [l+1, r]. \qquad \begin{aligned} l \leq k < i &\implies A[k] \leq x \\ \land i \leq k \leq j &\implies A[k] > x \\ \land k = l &\implies A[k] = x \end{aligned}$$

  IMAGE HERE

- **Analysis of Partition**: Time complexity of $\Theta(n)$ where $n = (r+1) - l$

- **Analysis of Quicksort**: We have the recurrence relation

$$T(n) = T(p) + T(n-p-1) + \Theta(n).$$

  - **Best Case**. Occurs when $p = n/2$. Recurrence relation reduces to
$$T(n) = 2T(n/2) + \Theta(n).$$
    Hence $T(n) \in \Theta(n \log_2 n)$.
  - **Worst Case**. Occurs when list is sorted in reverse order. Hence
$$T(n) = T(n-1) + \Theta(n).$$
    Yielding $T(n) \in \Theta(n^2)$

3.5.1 Randomized Quicksort

- Randomized quicksort algorithm is used to avoid unbalanced partitioning.

- Swap $A[l]$ with a random element in the subarray $A[l : r+1]$.

- Running time is independent of the initial order of $A \implies$ no inputs elicit a worst case behavior.

- Expected running time is $O(n \log_2 n)$

3.6 Heapsort

Algorithm 8 Heapsort

```
procedure HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i ← A.length − 1 downto 1 do
        swap A[0], A[i]
        A.heap-size − −
        MAX-HEAPIFY(A)
    end for
end procedure
```

- **Correctness**: We prove the following loop invariant by induction on i:

$$\begin{aligned} P(i) = \ & \text{the subarray } A[: i+1] \text{ is a binary max-heap containg the } i \text{ smallest elements of } A \\ & \wedge \text{the sorted subarray } A[i :] \text{ contains the } n-i \text{ largest elements of } A \end{aligned}$$

- **Analysis**:

$$T(n) = \underbrace{O(n)}_{\text{building heap}} + (n-1)\underbrace{O(\log_2 n)}_{\text{heapify}} = O(n \log_2 n).$$

3.7 Sorting in Linear Time

3.7.1 Lower Bounds for Comparison Sorting

Theorem 3.7.1. Any comparison sort algorithm requires $\Omega(n \log_2 n)$ comparisons in the worst case.

- Comparison sorts on array A can be viewed as a decision tree
- A decision tree is a binary tree in which each vertex (i, j) is a comparisons between $A[i]$ and $A[j]$.
- Left subtree is comparisons given $A[i] \leq A[j]$ (vice versa).
- A leaf is a permutation σ of $[n]$. Hence $n!$ permutations $\implies$ $n!$ leaves.

- Execution of a sorting algorithm is a path from root to leaf, hence lower bound is given by the height h.

$$\begin{aligned} n! \leq \# \text{ of leaves } &= 2^h \\ \iff h &\geq \log_2 n! \\ \iff h &= \Omega(n \log_2 n) \end{aligned}$$

3.7.2 Counting Sort

- Sorts integer elements in a fixed range $[0, k]$.

Algorithm 9 Counting Sort

```
function COUNTING SORT(A, k)
    B ← array of length n
    C ← array of length k + 1
    for x ∈ A do
        C[x] + +
    end for
    for i ← 1 to k do
        C[i] ← C[i] + C[i − 1]
    end for
    for i ← A.length − 1 to 0 do
        B[C[A[i]] − 1] ← A[i]
        C[A[i]] − −
    end for
    return B
end function
```

- Idea:

    1. $C[i]$ stores the number of elements $\leq i$, so $C[A[i]] - 1$ stores the final position of $A[i]$
    2. Decrementing $C[A[i]]$ causes the next element equal to $A[i]$ to be placed before $C[A[i]]$

- **Analysis**: The complexity is $\Theta(n + k)$.

Definition 3.7.1. (**Stable Sorting Algorithm**) A sorting algorithm in which elements with the same value appear in the same order in the output as they do in the input.

3.7.3 Bucket Sort

- Sorting elements that are uniformly distributed over $[0, 1)$ (or any fixed range).

Algorithm 10 Bucket Sort

```
function BUCKET-SORT(A)
    B ← {{}} × n
    for i ← 0 to n − 1 do
        INSERT(B[⌊nA[i]⌋], A[i])
    end for
    for i ← 0 to n − 1 do
        INSERTION-SORT(B[i])
    end for
    return ⋃ B
end function
```

- Idea:
  1. Scale the range to $[0, 1)$.
  2. Create an array B of n lists. Each list $B[i]$ stores values between $[i/n, (i + 1)/n)$
  3. Sort each list and concatenate.
- **Analysis**:
$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2),$$
where n_i is the random variable denoting the length of list $B[i]$. The average time complexity is
$$\mathbb{E}[T(n)] = \Theta(n) + \sum_{i=0}^{n-1} O(\mathbb{E}[n_i^2]) = \Theta(n).$$
Proof follows from
$$n_i = \sum_{j=0}^{n-1} X_{ij},$$

where

$$X_{ij} = \begin{cases} 1 & \text{if } A[j] \in B[i] \\ 0 & \text{otherwise} \end{cases}.$$

- For non-uniform distributions, suppose X is the random variable for elements in A and f is the density function. Define $Y = F_X(x)$ where

  $$F_X(x) = P(X \leq x) = \int_0^x f(t)\,\mathrm{d}t\,.$$

  Y is uniformly distributed. A distribution (between 0 and 1) is uniform if and only if

  $$F_Y(y) = y.$$

3.7.4 Radix Sort

- Sorts an array of d digit numbers.

Algorithm 11 Radix Sort

```
procedure Radix-Sort(A)
    for i ← 0 to d − 1 do
        stable sort A on digit i
    end for
end procedure
```

- Idea:
  - Sorts the least significant digit using a stable sort.
  - Repeat for all digits, moving from least to most.
- Sort from most to least produces k partitions each iteration, hence requires more space.
- **Correctness**: Prove the following loop invariant by induction on i:

  $$P(i) = \forall x \in A.\text{the array of subsequences } x_{i,0} \text{ is sorted},$$

  where $x_{i,0}$ is the sequence of the ith to 0th (least significant) digit.
- **Analysis**: The time complexity is $\Theta(d(n+k))$ where k is the number of possible values each digit can take, if it uses a stable sort algorithm that takes $\Theta(n+k)$ time.

4 Data Structures

4.1 Elementary Data Structures

- A dynamic set S is a finite set that may be mutated / updated.
- Stores unique keys (often related to satellite data).
- **Operations**:
  - Two categories: queries (read) and mutations (write).
  - SEARCH(S, k). A query, given S and a key k, returns pointer to element x such that $x.k = k$ (or **null**).
  - INSERT(S, x). A mutation, that inserts element x into S.
  - DELETE(S, x). A mutation, that deletes element x from S. (Assumes that x is in S)
  - MINIMUM(S), MAXMIMUM(S).
  - SUCCESSOR(S, x), PREDECESSOR(S, x).

4.1.1 Stacks

- Last-in, first-out (LIFO) data structure.
- **Operations**:
  - IS-EMPTY(S). Returns where S is empty.
  - PUSH(S, x). A mutation, that inserts (or pushes) element x onto the top of S.
  - POP(S). A mutation, that removes the top element from S. Assumes that IS-EMPTY(S) = **false**.

- Peek(S). A query, returns a pointer to the top element of S. Assumes that Is-Empty(S) = **false**.

- **Implementation**:

  - Array S of length n, with attribute $S.top$. IMAGE HERE

Algorithm 12 Is-Empty. Complexity: $\Theta(1)$

function Is-Empty(S)
    return $S.top = 0$
end function

Algorithm 13 Push. Complexity: $\Theta(1)$

procedure Push(S, x)
    if $S.top = n - 1$ **then**
        error "overflow"
    end if
    $S[++S.top] \leftarrow x$
end procedure

Algorithm 14 Pop. Complexity: $\Theta(1)$

function Pop(S)
    if Is-Empty(S) **then**
        error "underflow"
    end if
    return $S[S.top--]$
end function

  - Attempting to pop from empty stack $\implies$ underflow. Attempting to push onto a full stack $\implies$ overflow.

4.1.2 Queues

- First-in, first-out (FIFO) data structure.
- A queue has a head and a tail
  IMAGE

- **Operations**:
  - IS-EMPTY(Q). Returns where Q is empty.
  - ENQUEUE(Q, x). A mutation, that inserts element x at the tail of Q.
  - DEQUEUE(Q). A mutation, that removes the element from the head of Q. Assumes that IS-EMPTY(Q) = **false**.
  - PEEK(Q). A query, returns a pointer to the head of Q. Assumes that IS-EMPTY(Q) = **false**.
- **Implementation**:
  - Array Q of length n, with attributes $Q.head$ and $Q.tail$.

Algorithm 15 Is-Empty. Complexity: $\Theta(1)$

function IS-EMPTY(Q)
    return $Q.head = Q.tail$
end function

Algorithm 16 Enqueue. Complexity: $\Theta(1)$

procedure ENQUEUE(Q, x)
    if SIZE(Q) $= n - 1$ **then**
        error "overflow"
    end if
    $Q[Q.tail] \leftarrow x$
    $Q.tail \leftarrow Q.tail + 1 \pmod n$
end procedure

Algorithm 17 Dequeue. Complexity: $\Theta(1)$

```
function DEQUEUE(Q)
    if IS-EMPTY(Q) then
        error "underflow"
    end if
    x ← Q[Q.head]
    Q.head ← Q.head − 1 (mod n)
    return x
end function
```

Deque

- A deque (doubly ended queue) is a queue variant with enqueue and dequeue operations at each end of the queue.

4.1.3 Linked Lists

- A linked list is a data structure where elements are arranged in a linear order, based on a sequence of pointers. IMAGE
- A doubly linked list, each node contains a pointer to the next and previous nodes in the sequence. IMAGE
- A circular list is where the next pointer of the tail points to $L.head$ (and the previous pointer of $L.head$ points to the tail). IMAGE
- **Operations**: All dynamic set operations from section ?? are supported by linked lists.
- **Implementation**:
  - A circular doubly linked list L with the attribute $L.nil$, a sentinel node used to avoid boundary conditions. IMAGE

Algorithm 18 Insert. Complexity: $\Theta(1)$

```
procedure INSERT(L, x)
    x.next ← L.nil.next
    L.nil.next.prev ← x
    x.prev ← L.nil
    L.nil.next ← x
end procedure
```

Algorithm 19 Delete. Complexity: $\Theta(1)$

```
procedure DELETE(L, x)
    x.prev.next ← x.next
    x.next.prev ← x.prev
end procedure
```

4.2 Hashing

- A hash table is a data structure that implements a dictionary abstract data type.
- A dictionary abstract data type is a dynamic set S with n slots that supports the operations:
  - $\textsc{Insert}(S, x)$: $S \leftarrow S \cup \{x\}$.
  - $\textsc{Delete}(S, x)$: $S \leftarrow S \setminus \{x\}$.
  - $\textsc{Search}(S, k)$: returns a pointer to the element x such that $x.k = k$ (or **null**).

4.2.1 Direct-Addressing

- Suppose keys k are from a universal set $\mathcal{U} = \{0, 1, \ldots, m-1\}$.
- Assume no two distinct elements x, y have the same key.

- Use an array $T[0 \dots m-1]$ (direct-address table) to represent the dynamic set S, where

$$T[k] = \begin{cases} x & \text{if } x \in S \wedge x.k = k \\ \textbf{null} & \text{otherwise} \end{cases} .$$

- **Implementation**:
  - INSERT(T, x): $T[x.k] \leftarrow x$.
  - DELETE(T, x): $T[x.k] \leftarrow$ **null**.
  - SEARCH(T, k): $T[k]$.
  - **Analysis**: $\Theta(1)$ complexity.
- **Problem**: If $\mathcal{U}$ is large, then T is large. It may be impractical / impossible to store T.

4.2.2 Hashing

- A hash function h maps keys into slots (positions) of the hash table $T[0 \dots m-1]$.

$$h : \mathcal{U} \to \{0, 1, \dots, m-1\} .$$

  IMAGE

- **Problem**: Since $m < |\mathcal{U}|$, then it follows that there exists two distinct keys with the same hash value.

  Definition 4.2.1. (**Collision**) A collision is the event when a hash function h hashes two distinct keys into the same slot.

4.2.3 Resolving Collisions

Chaining

- Elements that hash into the same slot are placed into a linked list. IMAGE
- **Implementation**:

- $\textsc{Insert}(T, x)$. $\textsc{Insert}(T[h(x.k)], x)$. Assumes that $x.k$ is not in $T[h(x.k)]$.
- $\textsc{Delete}(T, x)$. $\textsc{Delete}(T[h(x.k)], x)$. Assumes that x is in $T[h(x.k)]$.
- $\textsc{Search}(T, k)$. $\textsc{Search}(T[h(k)], k)$.
- **Analysis**: INSERT and DELETE have $\Theta(1)$ time complexity.
- **Analysis of Search**:
  * **Worst Case**: All n keys hash to the same slot, hence $\Theta(n)$ time complexity.

    Definition 4.2.2. (**Load Factor**) The load factor of a hash table with n keys and m slots is defined as

    $$\alpha = \frac{n}{m} = \text{avg. \# of keys per slot.}$$
  * **Average Case**: Let us assume that h satisfies the simple uniform hashing property, that is each key $k \in \mathcal{U}$ is equally like to be hashed into any of the m slots in T. Hence

    $$P(h(k) = j) = \frac{1}{m},$$

    for $k \in \mathcal{U}$, $0 \leq j \leq m - 1$. Hence the expected length of the list at slot j, denoted n_j, is

    $$\mathbb{E}[n_j] = \alpha.$$

    Hence the average search time is

    $$\Theta(\underbrace{1}_{\text{cost of hash function}} + \underbrace{\alpha}_{\text{cost of searching list}}).$$

    If $n = O(m)$, then the cost is $\Theta(1)$.

4.3 Binary Search Trees

Definition 4.3.1. (**Binary Tree**) A binary tree is a tree (a connected acyclic graph) where each vertex has at most 2 children.

Definition 4.3.2. (**Binary Search Tree**) A binary search tree is a binary tree that satisfies the **binary search property** IMAGE HERE

4.3.1 Traversals

Algorithm 20 Inorder Traversal. Complexity: $\Theta(|V|)$

- **procedure** INORDER-TRAVERSAL(x)
    - **if** $x \neq$ **null then**
        - INORDER-TRAVERSAL($x.l$)
        - OUTPUT($x.k$)
        - INORDER-TRAVERSAL($x.r$)
    - **end if**
- **end procedure**

- Inorder traversal traverses the keys of the BST in sorted order. Proof of correctness by induction on tree height h.

$$\begin{aligned} P(h) = {} & \text{INORDER-TRAVERSAL}(x) \text{ applied to tree } x \text{ of height } h \\ & \text{traverses keys in sorted order} \end{aligned}$$

- **Analysis**:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 0 \\ T(k) + T(n-k-1) + 1 & \text{otherwise} \end{cases}.$$

Hence $T(n) \in \Theta(n)$. Proof using substitution method.

4.3.2 Insertion

Algorithm 21 Insert. Complexity: $O(h)$

```
 1: procedure INSERT(T, z)
 2:     y ← null, x ← T.root
 3:     while x ≠ null do
 4:         y ← x
 5:         x ← x.l if z.k < x.k else x.r
 6:     end while
 7:     z.p ← y
 8:     if y = null then
 9:         T.root ← z
10:     else
11:         if z.k < y.k then
12:             y.l ← z
13:         else
14:             y.r ← z
15:         end if
16:     end if
17: end procedure
```

- Two stages:
  - **Search**. lines 2-6. Searches for the parent p such that $y.k \leq p.k$ or $y.k \geq p.k$.
  - **Insert**. lines 7-16. Inserts y such that y satisfies **binary search property**.
- Time complexity: $O(h)$ time for tree of height h.

4.3.3 Deletion

- Cases:
  (i) z has no children $\implies$ simply delete z
  (ii) z has one child $\implies$ delete z and z is replaced by z's child.
  (iii) z has two children:
    - find z's successor y (minimum of z's right subtree)
    - replace z with y

- delete y from $z.r$

IMAGE HERE

Algorithm 22 Delete. Complexity: $O(h)$

```
procedure Delete(T, z)
    if z.l = null then
        Transplant(T, z, z.r)
    else
        if z.r = null then
            Transplant(T, z, z.l)
        else
            y ← Minimum(z.r)
            z.k ← y.k
            Delete(T, y)
        end if
    end if
end procedure
```

- Time complexity: $O(h)$.

4.3.4 Searching

Algorithm 23 Search. Complexity: $O(h)$

```
function Search(x, k)
    if x = null ∨ x.k = k then
        return x
    end if
    if k < x.k then
        return Search(x.l, k)
    else
        return Search(x.r, k)
    end if
end function
```

- Time complexity: $O(h)$, vertices encountered form a path from root to vertex. Hence maximum length of path $= h$. IMAGE HERE

- SEARCH is tail recursive, so an iterative implementation is possible:

Algorithm 24 Iterative-Search. Complexity: $O(h)$

```
function ITERATIVE-SEARCH(x, k)
    while x ≠ null ∧ k ≠ x.k do
        x ← x.l if k < x.k else x.r
    end while
    return x
end function
```

4.3.5 Maximum and Minimum

Algorithm 25 Minimum

```
function MINIMUM(x)
    while x.l ≠ null do
        x ← x.l
    end while
    return x
end function
```

Algorithm 26 Maximum

```
function MAXIMUM(x)
    while x.r ≠ null do
        x ← x.r
    end while
    return x
end function
```

- Time complexity: $O(h)$
- Correctness follows directly from **binary search property**.

4.3.6 Predecessor and Successor

Definition 4.3.3. The successor of a vertex x is the vertex y with the *smallest greatest key* than $x.k$, that is

$$x.k < y.k \wedge \nexists z. x.k < z.k < y.k.$$

Algorithm 27 Successor. Complexity: $O(h)$

- **function** SUCCESSOR(x)
    if $x.r \neq$ **null then**
        return MINIMUM($x.r$)
    end if
    $y \leftarrow x.p$
    while $y \neq$ **null** $\wedge\, y.r = x$ **do**
        $x \leftarrow y$
        $y \leftarrow y.p$
    end while
    return y
end function

- Case $x.r =$ **null**. The successor y is the lowest ancestor of x whose left child is also an ancestor of x. IMAGE HERE Proof by contradiction, assume that y isn't the lowest ancestor of x.

4.4 Red Black Trees

Definition 4.4.1. (**Balanced Search Trees**) A search tree data structure maintaining a dynamic set of n elements with a tree of height $O(\log_2 n)$.

Definition 4.4.2. (**Red Black Trees**) A binary search tree that satisfies the **red-black properties**:

(i) Every vertex is either red of black.

(ii) The root and leaves (nils) are black.

(iii) If a vertex is red, then it's children are black.

(iv) All paths from a vertex x to the leaves of the subtree have the same # of black vertices $= bh(x)$ (black-height).

IMAGES

- Example:

Theorem 4.4.1. (**Height of a Red Black Tree**) A red black tree T with n vertices has height $\leq 2\log_2(n+1) = O(\log_2 n)$.

Proof. Convert T into a 2-3-4 tree using isomorphisms defined in section ?? e.g

IMAGE HERE

By property (??) of 2-3-4 trees, every leaf (nil) has the same depth (namely, $h' = bh(root)$ by (iv)). In a 2-3-4 tree, the # leaves satisfies

$$2^{h'} \leq \# \text{ leaves } \leq 4^{h'}.$$

Recall that the number of leaves in a binary tree with n vertices is $n+1$. So we have

$$\begin{aligned} 2^{h'} &\leq n+1 \\ \iff h' &\leq \log_2(n+1) \end{aligned}$$

Note that $h \leq 2h'$ (since at most 1/2 of the vertices on any root to leaf path are red). Hence

$$h \leq 2\log_2(n+1).$$

□

Corollary 4.4.1.1. A red black tree is a balanced search tree.

4.4.1 Rotations

IMAGE

- Preserves the **binary search property**.

$$\forall a \in \alpha, b \in \beta, c \in \gamma . a.k \leq x.k \leq b.k \leq y.k \leq c.k.$$

- Allows for rebalancing of the tree structure while preserving **binary search property**
- Time complexity: $\Theta(1)$.

4.4.2 Insertion

- Outline:
    1. Insert z into T using binary search tree procedure.

2. Color z red.
3. **Problem**: z's parent might be red (violated property (iii)).
4. Move violation of (iii) up the tree via recoloring until we can fix the violation using recoloring and rotations.

Algorithm 28 Insert. Complexity: $O(h)$

-
```
procedure RB-Insert(T, z)
    BST-Insert(T, z)
    z.color ← Red
    while z.p.color = Red do
        if z.p = z.p.p.l then (A)
            y ← z.p.p.r
            if y.color = Red then
                Case (i)
            else
                if z = z.p then
                    Case (ii)
                end if
                Case (iii)
            end if
        else (B)
            Same as (A) but reversing left ↔ right
        end if
    end while
    T.root.color ← Black
end procedure
```

- IMAGES
- **Cases**:
  (i) IMAGE New $z = z.p.p$ since possible violation of (iii).
  (ii) (Triangle) IMAGE New $z = z.p$ since violation of (iii).
  (iii) (Line) IMAGE No violations.

4.5 B-Trees

Definition 4.5.1. (**B-Tree**) A B-Tree T is a rooted tree satisfying:

(i) For all vertices $x \in T$, x contains

- $x.n$ is the # of keys in x
- keys of $x.k_i$ are sorted.
- $x.leaf$

(ii) For all internal vertices $x \in T$, x contains list of $x.n + 1$ pointers to children $x.c_i$.

(iii) The keys $x.k_i$ separate the children: Let k_i be any key in $x.c_i$

$$k_1 \leq x.k_1 \leq \cdots \leq x.k_n \leq k_{n+1}.$$

(iv) All leaves have the same depth h (the trees height).

(v) $t \geq 2$ is the minimum degree. All vertices (except $T.root$) have $t-1$ keys. All vertices have at most $2t-1$ keys.

Theorem 4.5.1. For n-key B-Tree of height h and minimum degree $t \geq 2$.

$$h \leq \log_t \frac{n+1}{2}.$$

Proof. IMAGE So we have

$$\begin{aligned} n &\geq \underbrace{1}_{\text{root}} + \underbrace{(t-1)}_{\text{min \# of keys}} \cdot \sum_{k=1}^{h} 2t^{k-1} \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &= 2t^h - 1 \end{aligned}$$

Hence

$$t^h \leq \frac{n+1}{2} \implies h \leq \log_t \frac{n+1}{2}.$$

□

4.5.1 Insertion

- **Steps**:
  - Traverse down until leaf. Insert in leaf.
  - If visit full vertex while traversing, then split. Ensures we can insert into parent if leaf is full.
- **Splitting**: IMAGE

Algorithm 29 Split. Complexity: $O(t)$

- **procedure** SPLIT(x, i)
  - $y \leftarrow x.c_i$, $z \leftarrow$ **new vertex**
  - $z.k[:] \leftarrow y.k[t+1 : 2t], z.c[:] \leftarrow y.c[t+1 : 2t+1]$
  - $k_t \leftarrow y.k_t, y.n \leftarrow t-1$
  - $y.k[:] \leftarrow y.k[0 : t]$
  - INSERT-AT$(x.k, k_t, t)$
  - $x.n++$
- **end procedure**

Algorithm 30 Insert. Complexity: $O(t \log_t n)$

- **procedure** INSERT-NONFULL(x, k)
  - **if** $x.leaf$ **then**
    - *Insert k into x*
  - **else**
    - *Find i s.t. $k \leq x.k_i$*
    - **if** $x.c.n_i = 2t-1$ **then**
      - SPLIT(x, i)
      - $i \leftarrow i+1$ **if** $k > x.k_i$
    - **end if**
    - INSERT-NONFULL$(x.c_i, k)$
  - **end if**
- **end procedure**

- x must be non-full. If root is full, then split before calling INSERT-NONFULL.

- **Analysis**: $\Theta(ht) = \Theta(t \log_t n)$ since INSERT-NONFULL recurses h times by property (iv).

4.5.2 Deletion

- **Steps** (Cases):
  - If $k \notin x.k$.
    1. Determine subtree $x.c_i$ that contains k.
    2. If $x.c_i$ contains $t-1$ keys:
       (a) If $x.c_{i+1}$ (or $x.c_{i-1}$) have t keys, move $x.c_{i+1}.k_1$ into x. Move $x.k_i$ into $x.c_i$.
       (b) If $x.c_{i+1}$ (and $x.c_{i-1}$) have $t-1$ keys. **Merge** $x.c_i$ and $x.c_{i+1}$
    3. Recurse to $x.c_i$
  - If $k \in x.k$ and $\neg x.leaf$.
    1. If child y precedes k and has t keys, find predecessor k' of k. Swap k and k'. Recursively delete k from y.
    2. If child z succeeds k and has t keys, ...
    3. If y and z have $t-1$ keys. **Merge** k and all of z into y. y now has $2t-1$ keys. Recursively delete k from y.
  - If $k \in x.k$ and $x.leaf$, then delete k from x.

4.6 Binary Heaps

Definition 4.6.1. (**Binary Heap**) A binary heap is a binary tree with the following properties:

1. The binary tree is **complete**, every level except the bottom is completely filled.

2. The binary tree obeys a **heap property** .

Definition 4.6.2. (**Min Heap Property**) For heap H, H is said to object the min-heap property iff for all non-root vertices $x \in H$

$$x.parent.key \leq x.key.$$

Definition 4.6.3. (**Min Heap Property**) For heap H, H is said to object the max-heap property iff for all non-root vertices $x \in H$

$$x.parent.key \geq x.key.$$

- Binary heap of size $A.heap\text{-}size$ represented using an array A of length $A.length$.

$$\begin{aligned} parent(i) &= \lfloor (i+1)/2 \rfloor - 1 \\ left(i) &= 2i + 1 \\ right(i) &= 2i + 2 \end{aligned}$$

- Heapify is used to maintain heap property.

Algorithm 31 Max-Heapify. Complexity: $O(\log_2 n)$

procedure Max-Heapify(A, i)
  $l \leftarrow left(i), r \leftarrow right(i)$
  $largest \leftarrow l$ **if** $l \leq A.heap\text{-}size \wedge A[l] > A[i]$ **else** i
  $largest \leftarrow r$ **if** $r \leq A.heap\text{-}size \wedge A[r] > A[largest]$ **else** $largest$
  if $largest \neq i$ **then**
    swap $A[i], A[largest]$
    Max-Heapify$(A, largest)$
  end if
end procedure

- **Correctness**: Proof by induction on h.

- **Analysis**: Worst case: when bottom level of the tree is exactly half full:

$$\begin{aligned} n &= \underbrace{2^{h-1} - 1}_{left} + \underbrace{2^{h-2} - 1}_{right} + 1 \\ &= 3 \cdot 2^{h-2} - 1 \end{aligned}$$

So

$$2^{h-1} - 1 = \frac{2}{3}(n+1) - 1 < \frac{2}{3}n$$

So we have the recurrence relation

$$T(n) \leq T\left(\frac{2}{3}n\right) + \Theta(1).$$

By the Master theorem: $T(n) \in O(\log_2 n)$.

- Building a heap in linear time:

Algorithm 32 Build-Max-Heap. Complexity: $O(n)$

```
procedure BUILD-MAX-HEAP(A)
    A.heap-size ← A.length
    for i ← parent(A.length) to 0 do
        MAX-HEAPIFY(A, i)
    end for
end procedure
```

- **Correctness**: Proof by induction on i with loop invariant

  $P(i)$ = At start of each iteration, vertices $i{+}1, \ldots, n$ is root of a max-heap.

- **Analysis**:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \log_2 n \rfloor} \underbrace{\left\lceil \frac{n}{2^{h+1}} \right\rceil}_{\#\text{ vertices at height } h} O(h) \\ &= O\left(n \sum_{h=0}^{\log_2 n} \frac{h}{2^h}\right) \\ &= O\left(n \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^h}}_{2}\right) \\ &= O(n) \end{aligned}$$

- Can be used to implement a priority queue S:

- INSERT(A, x). $A = A \cup \{x\}$. Add element to end of the heap and bubble it up. $O(\log_2 n)$.
- MAXIMUM(A). Return element of S w/ largest key. $\Theta(1)$.
- EXTRACT-MAX(A). Remove and returns element of S w/ largest key. Read maximum, replace with last element and then heapify. $O(\log_2 n)$
- INCREASE-KEY(A, i, k). Increase key and then bubble up. $O(\log_2 n)$.
- DELETE(A, i). $O(\log_2 n)$
  * Decrease key of $A[i]$ to $-\infty$.
  * Then perform EXTRACT-MIN(A).

- UNION:
  - Two methods. Either append arrays and heapify $O(n)$,
  - or extract and insert $O(n \log_2 n)$.

4.7 Binomial Heaps

Definition 4.7.1. (**Binomial Heap**) A binomial heap H is a set of binomial trees S with the following properties:

- For all binomial trees $T \in S$. T obeys the min-heap property.
- There can only be zero or one binomial trees of order k in H.

Definition 4.7.2. (**Binomial Tree**) A binomial tree B_k is a rooted tree recursively defined as follows:

- B_0 is a single vertex.
- B_k, $k \geq 1$ is a root whose children are binomial trees $B_{k-1}, \ldots, B_0$. or combining two trees of order $k-1$.

IMAGE

- Properties of B_k (by induction on k):
  - 2^k vertices in B_k

  - Height of B_k is k
  - Exactly $\binom{k}{i}$ vertices at depth $0 \leq i \leq k$.
  - Root has degree k.

- If H has n vertices, then H contains $\lceil \log_2 n \rceil$ binomial trees. Proved by converting n into binary representation. If $n.b_k = 1$ then $B_k \in S$. $n.b$ requires at most $\lceil \log_2 n \rceil$ bits $\implies$ $\lceil \log_2 n \rceil$ binomial trees.

- Operations:

  - Insert(H, x). Finding minimum m s.t that $B_m \notin S$. If $m = 0$, then create B_0. Otherwise create B_m using $x, B_0, \ldots, B_{m-1}$. $O(\log_2 n)$
  - Minimum(H). Find the root B_m w/ min value. $O(\log_2 n)$.
  - Extract-Min(H). $O(\log_2 n)$
    - Find the root B_m with minimum value.
    - Remove from tree. It's children $B_0, \ldots, B_{m-1}$ form the binomial heap H'.
    - Union H' with H.
  - Decrease-Key(H, x, k'). $O(\log_2 n)$
    - Decrease key of vertex x.
    - Swap x with parent until min-heap property is satisfied.
  - Union(H, H'). $O(\log_2 n)$
    - Start from order 0 at stop at maximum order of H, H'.
    - At each order combine the trees from that order from the two heaps.
  - Delete(H, x). $O(\log_2 n)$
    - Decrease key of x to $-\infty$.
    - Then perform Extract-Min(H).

4.8 Fibonacci Heaps

Definition 4.8.1. (**Fibonacci Heap**) A fibonacci heap H is a collection of rooted trees that satisfy the min-heap property.

- Vertex structure of x:
  - $x.p$: pointer to parent.
  - $x.c_i$: pointers to children of x. Stored in a doubly-linked list.
  - $x.left$ and $x.right$ are pointers to x's siblings.
  - $x.degree = \#$ of children.
  - $x.loser$ is true if x has lost a child.
- Structure of fib heap H:
  - $H.min$ is a pointer to root of tree containing minimum key.
  - $H.roots$ pointer to root list (circular doubly linked list).
  - $H.n = \#$ of vertices in H.
- Operations:
  - Insert(H, x).
    - Initialize structure of x.
    - Insert x into $H.roots$.
    - Update $H.n$ and $H.min$ (if necessary)
  - Minimum(H).
    - Return $H.min$
  - Extract-Min(H).

```
function Extract-Min(H)
    z ← H.min, H.roots.delete(z)
    for c ∈ z.children do
        H.roots.insert(c), c.p ← null
    end for
    while two roots of same degree do
        Merge them, maintain min-heap property
    end while
    Update H.min
    return z
end function
```

  - Decrease-Key(H, x, k).

```
procedure DECREASE-KEY(H, x, k)
    x.k ← k, y ← x.p.
    if y ≠ null ∧ x.k < y.k then
        CUT(H, x, y), CASCADING-CUT(H, y)
        Update H.min
    end if
end procedure
procedure CUT(H, x, y)
    y.children.delete(x), H.roots.insert(x).
    x.p ← null, x.loser ← false
end procedure
procedure CASCADING-CUT(H, y)
    z ← y.p
    if z ≠ null then
        if y.loser = false then
            y.loser ← true
        else
            CUT(H, y, z), CASCADING-CUT(H, z)
        end if
    end if
end procedure
```

  * $x.loser$ attribute used to minimize d_{max}, keeping EXTRACT-MIN fast.
  * $x.loser =$ **true** if x has lost a child. If x loses two children, cut x.

- UNION(H_1, H_2).
  * Create empty heap H.
  * $H.roots \leftarrow$ **concat** $H_1.roots, H_2.roots$.
  * $H.min \leftarrow \min\{H_1.min, H_2.min\}$, $H.n \leftarrow H_1.n + H_2.n$.
  * Return H
- DELETE(H, x).
  * Decrease key of x to $-\infty$.
  * Then perform EXTRACT-MIN(H).

4.8.1 Analysis

- Define potential function

$$\Phi(H) = \text{number of roots in } H + 2 \cdot (\text{number of loser vertices in } H)\,.$$

- $r(H) = \#$ of roots in H, $\ell(H) = \#$ of losers
- Insert(H, x):
  - True cost is $\Theta(1)$.

$$\begin{aligned}\hat{c} &= \Theta(1) + \Phi(H') - \Phi(H)\\ &= \Theta(1) + (r(H) + 1 + 2\ell(H)) - (r(H) + 2\ell(H))\\ &= \Theta(1)\end{aligned}$$

- Extract-Min(H):
  - d_{max} is maximum degree.
  - Two parts:
    1. Promote children of min to root list and demark:
       * Takes $O(d_{max})$ work.
       * Yields $O(r(H) + d_{max})$ trees.
    2. Merge trees M trees: $O(M)$
       * $O(M)$ work merging trees. $r'(H) = O(r(H) + d_{max}) - M$
       * $O(r(H'))$ work updating $H.min$. $r(H') \leq d_{max} + 1$
  - True cost is $O(M + d_{max})$.

$$\begin{aligned}\hat{c} &= O(M + d_{max}) + \Phi(H') - \Phi(H)\\ &= O(M + d_{max}) + (r(H') - 2\ell(H)) - (r(H) - 2\ell(H))\\ &= O(M + d_{max}) + O(r(H) + d_{max}) - M - r(H) = O(d_{max})\end{aligned}$$

- Decrease-Key(H, x, k):
  - **Case** new key doesn't violate min-heap property. $\Theta(1)$.

$$\hat{c} = \Theta(1) + \Phi(H') - \Phi(H) = \Theta(1)$$

- **Case** violation:
  - x is cut in $\Theta(1)$ time.
  - Suppose x has $\mathcal{L}$ loser ancestors. Takes $O(\mathcal{L})$ cutting these.
  - Eventually reach non-loser ancestor y. Marking y as loser takes $\Theta(1)$ time.
  - So $O(\mathcal{L})$ true cost.

$$\begin{aligned}\hat{c} &= O(\mathcal{L}) + \{(r(H) + \mathcal{L} + 1) + 2\,(\ell(H) - (\mathcal{L} + 1) + 1)\} - (r(H) + 2\ell(H)) \\ &= O(\mathcal{L}) + 1 - \mathcal{L} = O(\mathcal{L}).\end{aligned}$$

Theorem 4.8.1. Let x be a vertex in fib heap H and $x.degree = k$. Let $y_1, \ldots, y_k$ be the children of x (in order of merging). Then

$$\begin{aligned} y_1.degree &\geq 0 \\ y_i.degree &\geq i - 2 \end{aligned}$$

Proof. IMAGES Trivially $y_1.degree \geq 0$. For $i \geq 2$ at the time x and y_i were merged, $x.degree = y_i.degree = i - 1$. So $y_i.degree \geq i - 1$. Since y_i could since be marked as a loser by Decrease-Key, implying it lost at most one child, hence $y_i.degree \geq i - 2$. □

Theorem 4.8.2. Let x be a vertex in fib heap H w/ degree k. Let N_k be the minimum # of vertices in tree rooted at x.

$$N_k \geq F_{k+2} \geq \phi^k.$$

Proof. We have

$$\begin{aligned} N_k &= \underbrace{1}_{x} + \underbrace{N_0}_{y_1} + \underbrace{N_0}_{y_2} + N_1 + \cdots + \underbrace{N_{k-2}}_{y_k} \\ &= 2 + \sum_{i=0}^{k-2} N_i \end{aligned}$$

Note that instantiating the above with $k-1$ yields

$$N_{k-1} = 2 + \sum_{i=0}^{k-3} N_i$$

So

$$N_k - N_{k-1} = N_{k-2} \implies N_k = N_{k-2} + N_{k-1}.$$

Hence N_k has the defining equation of the Fibonacci sequence with the base case $N_0 = 1$ and $N_1 = 2$. (as opposed to $F_0 = 0$ and $F_1 = 1$). So we have $N_k = F_{k+2}$ $\square$

4.9 Disjoint Sets

Definition 4.9.1. (**Disjoint-Set**) A disjoint set $\mathcal{S}$ is a collection of disjoint sets $\mathcal{S} = \{S_1, \ldots, S_n\}$

- Operations:
  - BUILD-SET(x). Creates a new set S in $\mathcal{S}$ only containing x.
  - UNION(x, y). Performs $S_x \cup S_y$. Conceptually this removes S_x, S_y from $\mathcal{S}$.
  - FIND-SET(x). Returns pointer to the set S_x containing x.

4.9.1 Linked List Representation

- Disjoint set can be represented using a collection of linked-lists $\mathcal{L}$.
- A singly linked list L is a set. Each element $x \in L$ has a pointer $x.set$ to L.
- BUILD-SET(x). Initializes x and L, points $L.head$ to x, returns L. $\Theta(1)$.
- UNION(x, y). Appends the elements of $y.set$ (S_y) to $x.set$ (S_x). Each element in $z \in S_y$ must have $z.set$ updated $\implies$ amortized $\Theta(n)$ time.
- FIND-SET(x). Returns $x.set$. $\Theta(1)$ time.
- **Weighted Union Heuristic**: Store the length of each list. When performing UNION append the shorter list onto the longer list.

Theorem 4.9.1. Using a linked-list implementation and weighted-union, a sequence of m BUILD-SET, UNION and FIND-SET where $n \leq m$ operations are BUILD-SET takes $O(m + n \log_2 n)$.

Proof.

- n disjoint sets, we must perform at most $n-1$ UNION operations.
- Consider upper bound for time taken by UNIONs.
- Let x be arbitrary. First time x is updated is in list of size 1, then 2, then 4, ...
- So for all $k \leq n$, after x has been updated $\lceil \log_2 k \rceil$ times, x is in a list of k elements. So x has been updated $\lceil \log_2 n \rceil$ times after in final list of n elements.
- So the aggregate for n elements $\implies O(n \log_2 n)$ time.
- Each BUILD-SET, MAKE-SET(o)peration takes $\Theta(1)$ time, hence $O(m)$ time spent.
- So total time is $O(m + n \log_2 n)$.

□

4.9.2 Forest Representation

- Disjoint set represented using a forest $\mathcal{F}$, a collection of trees.
- FIND-SET operation traverses the tree back to the root of the set. The path is known a **find path**.
- Two heuristics:
  - **Union by Rank**:
    * Store the upper bound for height, the **rank** of the tree.
    * UNION(x, y). For two trees t_1, t_2 with ranks r_1, r_2. If $r_1 < r_2$, then t_1 becomes the child of t_2. (vice-versa). Resulting rank is

      $$\text{Resulting rank} = \begin{cases} \max\{r_1, r_2\} & r_1 \neq r_2 \\ r_1 + 1 & r_1 = r_2 \end{cases}.$$

      Only requires updating a single pointer $\implies \Theta(1)$ time.
    * Aggregate analysis, m operations on n elements $O(m \log_2 n)$.

- **Path Compression**:
  - During FIND-SET operations, make each vertex in the **find path** point directly to the root.
  - Doesn't affect rank since rank is an **upper bound** of height.
  - Aggregate analysis, m operations on n elements $O(m\alpha(n))$.
    - $\alpha(n)$ grows **very** slowly and can be ignored.
    - This yields an amortized $O(1)$ time complexity per operation.

5 Amortized Analysis

- Amortized analysis is the analysis of a sequence of n operations with costs $c_1, \ldots, c_n$. The **amortized costs** $\hat{c_1}, \ldots, \hat{c_n}$ satisfy the fundamental inequality of amortized analysis

$$\forall j \leq n. \sum_{i=1}^{j} c_i \leq \sum_{i=1}^{j} \hat{c_i}$$

aggregate true cost $\leq$ aggregate amortized cost

- **Goal**: The goal of amortized analysis is to show that the average cost per operation is small, despite some operations being expensive.
- Types of amortized arguments:
  - Aggregate analysis
  - Accounting method (non-examinable)
  - Potential method

5.1 Aggregate Analysis

- Aggregate analysis consists of determining the worst-case upperbound $T(n)$ on the cost of n operations, then calculates the amortized cost to be $T(n)/n$.

Example 5.1.1. (**Dynamic Array**)

Definition 5.1.1. (**Dynamic Array**) A dynamic array is a list abstract data type implementation, using a maintained fixed length array; which when full allocates a new array with double the capacity and copies across the contents.

- **Analysis of Insert**:
  - The cost of doubling the capacity from m to $2m$ and copying is $\Theta(m)$.
  - Hence the worst-case upperbound of n insert operations is

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lfloor \log_2 n \rfloor} \Theta(2^i) \\ &= \Theta(\frac{2n-1}{2-1}) \\ &= \Theta(n) \end{aligned}$$

  - So the amortized cost of insert is

$$\hat{c}_{insert} = \frac{\Theta(n)}{n} = \Theta(1).$$

5.2 Potential Method

- Amortized analysis using the potential method consists of defining a potential function $\Phi(S)$ for the "potential" of a data structure of state S.
  - Can release potential to "pay" for future operations.
  - Most flexible of the amortized analysis methods.
- Let

$$\begin{aligned} S_i &\text{ be the state of the data structure } D \text{ after the } i\text{th operation} \\ S_0 &\text{ be the initial state of the data structure } D \\ c_i &\text{ be the cost of the } i\text{th operation} \\ \hat{c}_i &\text{ be the amortized cost of the } i\text{th operation} \end{aligned}$$

Define a potential function $\Phi : \{S_i\} \to \mathbb{R}$, such that

$$\begin{aligned} \Phi(S_0) &= 0 \\ \Phi(S_i) &\geq 0 \end{aligned}$$

for all $S_i \in \{S_i\}$, and

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(S_i) - \Phi(S_{i-1}) \\ &= c_i + \underbrace{\Delta\Phi(S_i)}_{\text{change in potential due to } i\text{th operation}}\end{aligned}$$

- If $\Delta\Phi(S_i) \geq 0$, then $\hat{c}_i \geq c_i$, so the ith operation does "work" on the data structure to pay for future operations.
- If $\Delta\Phi(S_i) < 0$, then $\hat{c}_i < c_i$, so the data structure does "work" to help pay for the ith operation.
- Consider sequence of operations with states and costs

$$S_0 \xrightarrow{c_1} S_1 \xrightarrow{c_2} S_2 \xrightarrow{c_3} \cdots \xrightarrow{c_n} S_n,$$

then we have

$$\begin{aligned}\text{Aggregate amortized cost} = \sum_{i=1}^{n} \hat{c}_i &= [c_1 + \Phi(S_1) - \Phi(S_0)] \\ &+ [c_2 + \Phi(S_2) - \Phi(S_1)] \\ &\vdots \\ &+ [c_n + \Phi(S_n) - \Phi(S_{n-1})] \\ &= \sum_{i=1}^{n} c_i + \Phi(S_n) - \Phi(S_0)\end{aligned}$$

So

$$\begin{aligned}\text{Aggregate true cost} = \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c}_i + \underbrace{\Phi(S_0)}_{0} - \underbrace{\Phi(S_n)}_{\geq 0} \\ &\leq \sum_{i=1}^{n} \hat{c}_i = \text{Aggregate amortized cost}\end{aligned}$$

Hence the potential method satisfies fundamental inequality of amortized analysis.

Example 5.2.1. (**Dynamic Array**)

- **Analysis Of Insert**:
  - Consider a dynamic array A. We define
$$\Phi(A) = 2 \cdot A.size - A.capacity.$$
  - For an empty array
$$\Phi(A) = 0.$$
    For all A
$$\begin{aligned} A.capacity \geq A.size &\geq \frac{1}{2} A.capacity \\ \iff 2 \cdot A.size &\geq A.capacity \end{aligned}$$
    Hence $\Phi(A) \geq 0$.
  - Cases:
    * **Case** $A.size \leq A.capacity$ (No expansion): We have
$$\begin{aligned} A_i.capacity &= A_{i-1}.capacity \\ A_i.size &= A_{i-1}.size + 1 \\ c_i &= \Theta(1) \end{aligned}$$
      So
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(A_i) - \Phi(A_{i-1}) \\ &= \Theta(1) + [2 \cdot (A_{i-1}.size + 1) - A_{i-1}.capacity] - [2 \cdot A_{i-1}.size - A_{i-1}.capacity] \\ &= \Theta(1) \end{aligned}$$
    * **Case** $A.size > A.capacity$ (Expansion): We have
$$\begin{aligned} A_i.capacity &= 2 \cdot A_{i-1}.capacity \\ A_i.size &= A_{i-1}.capacity + 1 \\ c_i &= \Theta(A_i.size) \end{aligned}$$
      So
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(A_i) - \Phi(A_{i-1}) \\ &= \Theta(A_i.size) + [2 \cdot (A_{i-1}.capacity + 1) - 2 \cdot A_{i-1}.capacity] \\ &\quad - [2 \cdot (A_i.size - 1) - (A_i.size - 1)] \\ &= \Theta(A_i.size) + 2 - (A_i.size - 1) \\ &= \Theta(1) \end{aligned}$$

6 Graphs