DSAA(H) Review Sheet

Lecture 1

Algorithm

An algorithm is a well-defined **computational procedure** that takes some **input** and produces some **output**.

- It is a tool for solving a well-specified **computational problem**.

Insertion-Sort

```
INSERTIONSORT(A)
1: for j = 2 to A.length do
2:     key = A[j]
3:     // Insert A[j] into the sorted sequence A[1 . . . j − 1].
4:     i = j − 1
5:     while i > 0 and A[i] > key do
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

Correctness Proof: Loop invariant (Example)

```
INSERT-ALL-FIVES(A, n)
1: for i = 1 to n do
2:     A[i] = 5
```

Loop invariant: At the start of each iteration i of the loop, each element of the subarray $A[1..i-1]$ is 5.

• **Initialization:** For $i = 1$ the empty subarray has no elements (trivial).

• **Maintenance:** Loop invariant says that at step i of the *for* loop the subarray $A[1..i-1]$ contains 5s. During the i iteration we insert a 5 in $A[i]$, so by the end of the iteration the loop invariant still holds for step $i+1$.

• **Termination:** The algorithm terminates when $i = n+1$. Then the loop invariant for $i = n+1$ says that all the elements of the subarray $A[1..n]$ contain 5s, so the algorithm returns the correct output.

➢ Correctness of InsertionSort

- **Loop invariant:** *"At the start of each iteration of the for loop of lines 1-8, the subarray A[1..j-1] consists of the elements originally in A[1..j-1], but in sorted order."*

- **Initialisation:** For j=2 the subarray A[1] is the original A[1] and it is sorted (trivially).

- **Maintenance:** The while loop moves A[j-1], A[j-2], … one position to the right and inserts A[j] at the correct position i+1. Then A[1..j] contains the original A[1..j], but in sorted order:

$$\underbrace{A[1] \leq A[2] \leq \cdots \leq A[i-1] \leq A[i]}_{\text{sorted before}} \underbrace{\leq A[i+1] \leq}_{\text{from while loop}} \underbrace{A[i+2] \leq \cdots \leq A[j]}_{\text{sorted before}}$$

- **Termination:** The for loop ends when j=n+1. Then the loop invariant for j=n+1 says that the array contains the original A[1..n] in sorted order!

Other Basic Sorts

```
SELECTION-SORT(A)
1: n = A.length
2: for j = 1 to n − 1 do
3:     smallest = j
4:     for i = j + 1 to n do
5:         if A[i] < A[smallest] then smallest = i
6:     exchange A[j] with A[smallest]
```

```
BUBBLE-SORT(A)
1: for i = 1 to A.length − 1 do
2:     for j = A.length downto i + 1 do
3:         if A[j] < A[j − 1] then
4:             exchange A[j] with A[j − 1]
```

```
DO-I-SORT(A, n)
1: for i = 1 to n do
2:     for j = 1 to n do
3:         if A[i] < A[j] then
4:             exchange A[i] with A[j]
```

Lecture 2

Definition of $\Theta(g(n))$

$$\Theta(g(n)) = \{f(n): \text{there exist constants } 0 < c_1 \le c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$.

- $2n^2 - 10n = \Theta(n^2)$ since for all $n \ge n_0$
  $0 \le c_1 n^2 \le 2n^2 - 10n \le c_2 n^2$
  when choosing, say, $c_1 = 1, c_2 = 2, n_0 = 10$
  (as after division by n^2 we have $1 \le 2 - 10/n \le 2$ for $n \ge 10$)

Definition of $O(g(n)), \Omega(g(n)), o(g(n)), \omega(g(n))$

$$O(g(n)) = \{f(n): \text{there exist constants } 0 < c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}$$

$$\Omega(g(n)) = \{f(n): \text{there exist constants } 0 < c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}$$

$$f(n) = o(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \omega(g(n)) \text{ if } g(n) = o(f(n))$$

- For two non-negative functions *f(n), g(n)*:
  1. Slower functions can be ignored:
     $$f(n) + g(n) = \Theta(\max(f(n), g(n)))$$
  2. Asymptotic times can be multiplied:
     $$\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$$

Lecture 3

Merge-Sort

```
MERGESORT(A, p, r)
1: if p < r then
2:     q = ⌊(p + r)/2⌋
3:     MERGESORT(A, p, q)
4:     MERGESORT(A, q + 1, r)
5:     MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1: n_1 = q − p + 1                                          ┐
 2: n_2 = r − q                                              │
 3: let L[1 . . . n_1 + 1] and R[1 . . . n_2 + 1] be new arrays │ Set up arrays L
 4: for i = 1 to n_1 do                                      │ and R (boring)
 5:     L[i] = A[p + i − 1]                                  │
 6: for j = 1 to n_2 do                                      │
 7:     R[j] = A[q + j]                                      │
 8: L[n_1 + 1] = ∞                                           │
 9: R[n_2 + 1] = ∞                                           ┘
10: i = 1                                                    ┐
11: j = 1                                                    │
12: for k = p to r do                                        │
13:     if L[i] ≤ R[j] then                                  │ Actual merge
14:         A[k] = L[i]                                      │
15:         i = i + 1                                        │
16:     else                                                 │
17:         A[k] = R[j]                                      │
18:         j = j + 1                                        ┘
```

Master Theorem

Let *a > 0 and b > 1* be constants, and let *f(n)* be non-negative for large enough n. Then, the solution of the recurrence function defined over $n \in \mathbb{N}$

$$T(n) = a\,T(n/b) + f(n)$$

has the following asymptotic behaviour:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Lecture 4

- In place: $O(1)$ additional space.

Max-Heapify (assume maxheap)

```
MAX-HEAPIFY(A, i)
 1: l = Left(i)
 2: r = Right(i)
 3: if l ≤ A.heap-size and A[l] > A[i] then
 4:       largest = l
 5: else
 6:       largest = i
 7: if r ≤ A.heap-size and A[r] > A[largest] then
 8:       largest = r
 9: if largest ≠ i then
10:       exchange A[i] with A[largest]
11:       MAX-HEAPIFY(A, largest)
```

含义：将点 i 赋值为某个数之后，发现不符合堆的性质时与孩子节点做交换，向下扩展。

Building a heap

```
A.heap-size = n
for i = floor(n / 2) down to 1 do
    Max-Heapify(A, i)
```

Proof

- **Loop invariant**: At the start of each iteration i of the for loop, each node $i+1, i+2, \ldots, i+n$ is the root of a max-heap.
- **Initialization:** True for $i = \lfloor \frac{n}{2} \rfloor + 1, \cdots, n$.
- **Maintenance:** by loop invariant, all children of i are roots of max-heaps (as their numbers are larger than i). Then $\text{Max-Heapify}(A, i)$ turns the subtree at i into a max-heap.
- **Termination:** the loop terminates at $i = 0$, hence node 1 is the root of a max-heap.

Proof that building a heap is $O(n)$.

$$\sum_{h=1}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=1}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Heap-Sort

```
HEAPSORT(A)
1: BUILD-MAX-HEAP(A)
2: for i = A.length downto 2 do
3:     exchange A[1] with A[i]
4:     A.heap-size = A.heap-size −1
5:     MAX-HEAPIFY(A, 1)
```

Proof

- **Loop Invariant:** At the start of each iteration of the for loop of lines $2 \sim 5$, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the *n–i* largest elements of $A[1..n]$, sorted."
- **Initialization:** The subarray $A[i+1..n]$ is empty, thus the invariant holds.
- **Maintenance:** $A[1]$ is the largest element in $A[1..i]$ and it is smaller than the elements in $A[i+1..n]$. When we put it in the $i-$th position, then $A[i..n]$ contains the largest elements, sorted. Decreasing the heap size and calling Max-Heapify turns $A[1..i-1]$ into a max-heap. Decrementing i sets up the invariant for the next iteration.
- **Termination:** After the loop $i = 1$. This means that $A[2..n]$ is sorted and $A[1]$ is the smallest element in the array, which makes the array sorted.

Lecture 5

Quick-Sort

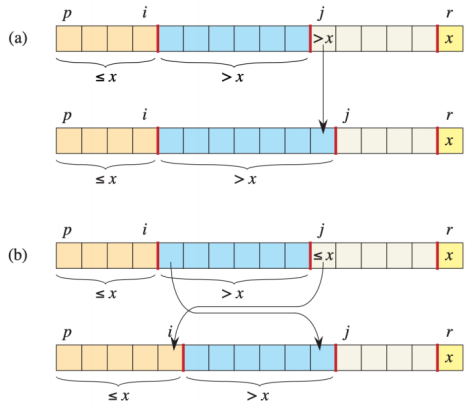
```
QUICKSORT(A, p, r)
1: if p < r then
2:     q = PARTITION(A, p, r)
3:     QUICKSORT(A, p, q − 1)
4:     QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1: x = A[r]
2: i = p − 1
3: for j = p to r − 1 do
4:     if A[j] ≤ x then
5:         i = i + 1
6:         exchange A[i] with A[j]
7: exchange A[i + 1] with A[r]
8: return i + 1
```

Proof of correctness:

- **Loop Invariant:** At the beginning of the j iteration: $A[p \ldots i] \leq x$ and $A[i+1, \ldots, j-1] > x$
- **Initialization:** Trivially true.
- **Maintenance:** If line 4 is false : picture (a); If line 4 true: picture (b). In both cases after one iteration of j the loop invariant is maintained.

- **Termination:** After the last swap in line 7, $A[p..i] \leq x < A[i+1..r]$, and Partition returns the position of x.



Runtime

- Worst: $\Theta(n^2)$.
- Best: $\Theta(n \log n)$.
- Average:

$$\begin{aligned} T(n) &= \frac{1}{n} \cdot \sum_{q=1}^{n} (T(q-1) + T(n-q) + \Theta(n)) \\ &= \frac{1}{n} \cdot \sum_{q=1}^{n} T(q-1) + \frac{1}{n} \cdot \sum_{q=1}^{n} T(n-q) + \frac{1}{n} \cdot \sum_{q=1}^{n} \Theta(n) \\ &= \frac{1}{n} \cdot \sum_{k=0}^{n-1} 2T(k) + \Theta(n) \end{aligned}$$

归纳证明：假设对 $0 \sim n-1$，$T(k) \leq c \cdot k \log k$. 那么有

$$\sum_{k=0}^{n-1} T(k) \approx c \sum_{k=2}^{n-1} k \log k \leq c \int_{2}^{n} x \log x \mathrm{d}x \approx \frac{cn^2}{2} \log n - \frac{cn^2}{4}.$$

$$T(n) \leq n + \frac{2}{n} c \left(\frac{n^2}{2} \log n - \frac{n^2}{4} \right) = n + cn \log n - \frac{cn}{2}$$

所以 $T(n) = O(n \log n)$.

Lecture 6

Randomized-Partition

$\textsc{Randomised-Partition}(A, p, r)$
1: $i = \textsc{Random}(p, r)$
2: exchange $A[r]$ with $A[i]$
3: **return** $\textsc{Partition}(A, p, r)$

Theorem: the **expected number of comparisons** of Randomized-QuickSort is $O(n \log n)$ for every input where all elements are distinct.

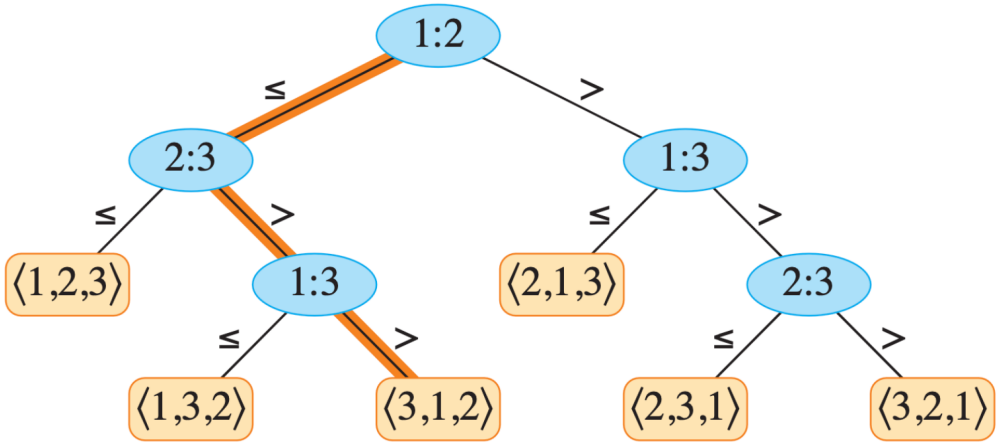
Lemma: The probability that z_i (i—th smallest) compared against z_j is $\frac{2}{j-i+1}$.

Therefore, substitute $k = j - i$, we have

$$E(X) = \sum_{j=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \leq 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \leq 2n \sum_{k=1}^{n} \frac{1}{k} = O(n \log n).$$

Comparison Sorts as Decision Trees

三个数的比较：



Theorem: Every comparison sort requires $\Omega(n \log n)$ comparisons in the worst case.

简单证明：决策树叶子节点有 $n!$ 个，所以最坏情况的比较次数至少是 $\log(n!)$ 的，而 $\log(n!) = \Omega(n \log n)$。

Lecture 7

Counting-Sort

```
CountingSort(A, B, k)
1:  let C[0...k] be a new array
2:  for i = 0 to k do
3:      C[i] = 0
4:  for j = 1 to A.length do
5:      C[A[j]] = C[A[j]] + 1
6:  for i = 1 to k do
7:      C[i] = C[i] + C[i − 1]
8:  for j = A.length downto 1 do
9:      B[C[A[j]]] = A[j]
10:     C[A[j]] = C[A[j]] − 1
```

Runtime: $\Theta(n+k)$. For the last loop:

- **Loop invariant:** At the start of each iteration j of the last for loop, the elements $A[j+1..n]$ are in the right position in B and the last element in A that has not yet been copied in B, with value $A[j] = i$, belongs to $B[C[i]]$."
- **Initialization:** At the start of the loop $j = n$ and no elements have been copied. The array C provides for each element, the number of elements in A that are smaller or equal to it. So the last element of A, $A[n] = i$, naturally goes in position $B[C[i]]$.
- **Maintenance:** At iteration j, the loop invariant tells us that the element $A[j] = i$ goes in $B[C[i]]$ and we copy it in. Since the next element equal to i in A that has not yet been copied in B should go in position $B[C[i-1]]$, we decrement $C[i]$ re-stablishing the loop invariant (the array C is updated such that each element i of A still to be copied in is indexed correctly again)
- **Termination:** When the loop terminates $j = 0$. The loop invariant tells us that all the elements of $A[1..n]$ are in the right position in B thus there are no more elements to be copied.

Radix Sort

```
RadixSort(A, d)
1: for i = 1 to d do
2:     use a stable sort to sort array A on digit i
```

Loop Invariant: At each iteration of the **for** loop, the array is sorted on the last $i-1$ digits.

Runtime: $\Theta(d(n+k))$, d is the number of base-k digits of elements.

Lecture 8

Stack & Queue (skip)

Linked-List

```
List-Search(L, k)
1: x = L.head
2: while x ≠ NIL and x.key ≠ k do
3:     x = x.next
4: return x
```

```
List-Prepend(L, x)
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4      L.head.prev =
5  L.head = x
```

```
List-Insert(x, y)
1  x.next = y.next
2  x.prev = y
3  if y.next ≠ NIL
4      y.next.prev = x
5  y.next = x
```

```
List-Delete(L, x)
1: if x.prev ≠ NIL then
2:     x.prev.next = x.next
3: else
4:     L.head = x.next
5: if x.next ≠ NIL then
6:     x.next.prev = x.prev
```

Lecture 9

Concepts of Trees

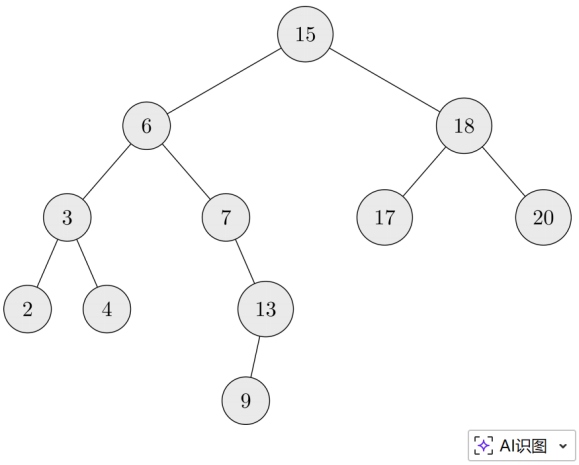
- A **path** in a tree is a sequence of nodes linked by edges.
- The **length** of a path is the number of edges.
- A **leaf** of a tree is a node that has no children; otherwise it is called **internal node**.
- We speak about **siblings, parents, ancestor, descendant** in the obvious way.
- The **depth** of a node in a tree is the length of a (simple) path from that node to the **root**.
- A **level** of a tree is a set of nodes of the same depth.
- The **height** of a node in a tree is the length of the **longest path** from that node to a leaf.

- A binary tree is **full** if each node is either a leaf or has exactly two children.
- It is **complete** if it is full and all leaves have the same level.

Theorem: A binary tree of height at most *h* has no more than 2^h leaves.

Binary Search Tree

- A binary search tree (BST) is a binary tree where all labels (keys) satisfy the **binary search tree property**:
  - If y is a node in the left subtree of x, then y.key ≤ x.key.
  - If y is a node in the right subtree of x, then y.key ≥ x.key.




```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2       return x
3  if k < x.key
4       return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2       if k < x.key
3            x = x.left
4       else x = x.right
5  return x
```

Runtime: *O(h), h* the height of the tree

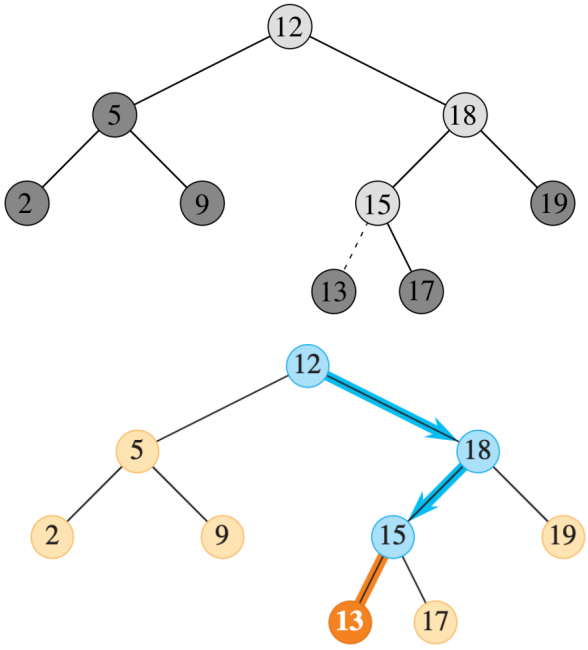
- **Minimum:** starting from the root, go left until the left child is NIL.
- **Maximum:** starting from the root, go right until the right child is NIL.

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)   // leftmost node in right subtree
3  else // find the lowest ancestor of x whose left child is an ancestor of x
4       y = x.p
5       while y ≠ NIL and x == y.right
6            x = y
7            y = y.p
8       return y
```

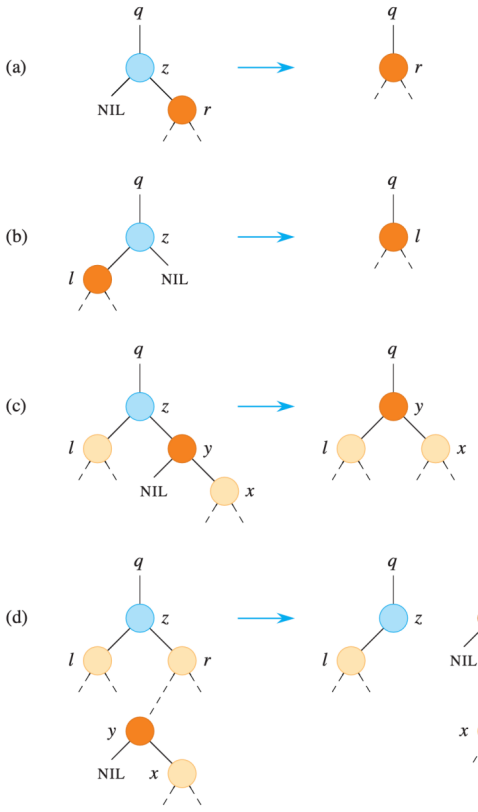
Insert(T,z)

```
TREE-INSERT(T, z)
1   x = T.root
2   y = NIL
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

Example: Insert(T,13)



Delete elements:



(a) Node has no children or only right child

(b) Node has only left child

(c) Special case where right child is the successor.

(d) Successor *y* is the minimum in right subtree; *y*'s left child is NIL. Swapping z and y.

Worst Runtime(one operation): $\Theta(n)$.

```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7       v.p = u.p
```

```
TREE-DELETE(T, z)
1   if z.left == NIL
2       TRANSPLANT(T, z, z.right)          // replace z by its right child
3   elseif z.right == NIL
4       TRANSPLANT(T, z, z.left)           // replace z by its left child
5   else y = TREE-MINIMUM(z.right)         // y is z's successor
6       if y ≠ z.right                     // is y farther down the tree?
7           TRANSPLANT(T, y, y.right)      // replace y by its right child
8           y.right = z.right              // z's right child becomes
9           y.right.p = y                  //       y's right child
10      TRANSPLANT(T, z, y)                // replace z by its successor y
11      y.left = z.left                    // and give z's left child to y,
12      y.left.p = y                       //       which had no left child
```

Tree Walks: Pre-order, In-order, Post-order.

Lecture 10

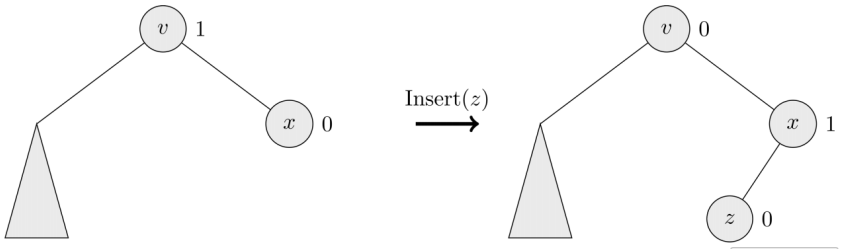
Theorem: The height of an AVL tree with n nodes is at most

$$h \leq \frac{1}{\log\left(\left(\sqrt{5}+1\right)/2\right)}\log n \approx 1.44 \log n$$

Recursion: $f(k) = f(k-1) + f(k-2) + 1 = \mathrm{Fib}(k) - 1$.

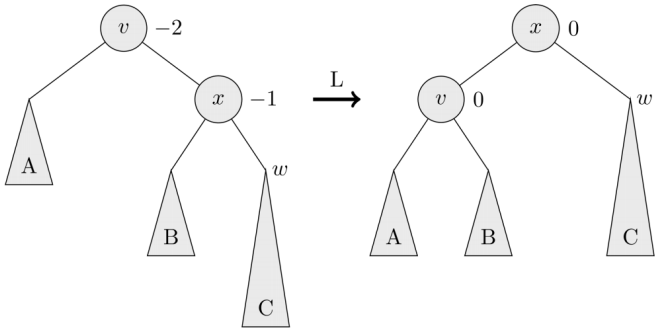
Insertion (negative bal means right subtree is deeper)

Case 1: $\mathrm{bal}(v) = 1 \to \mathrm{bal}(v) = 0$ **done.**

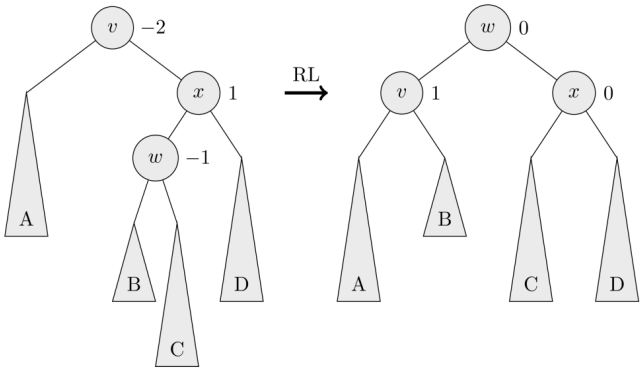


Case 2: $\mathrm{bal}(v) = 0 \to \mathrm{bal}(v) = -1$ **done.**

Case 3.1: $\mathrm{bal}(v) = -1 \to \mathrm{bal}(v) = -2$**, and** $\mathrm{bal}(v.\mathit{right}) < 0$



Case 3.2: $\mathrm{bal}(v) = -1 \to \mathrm{bal}(v) = -2$**, and** $\mathrm{bal}(v.\mathit{right}) > 0$



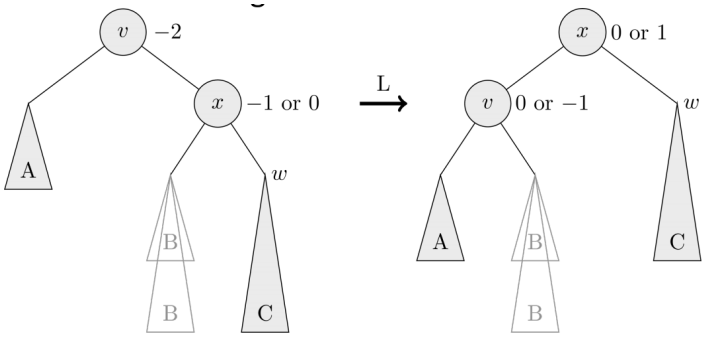
Other cases: symmetric of cases above.

Deletion (we list positive bal cases)

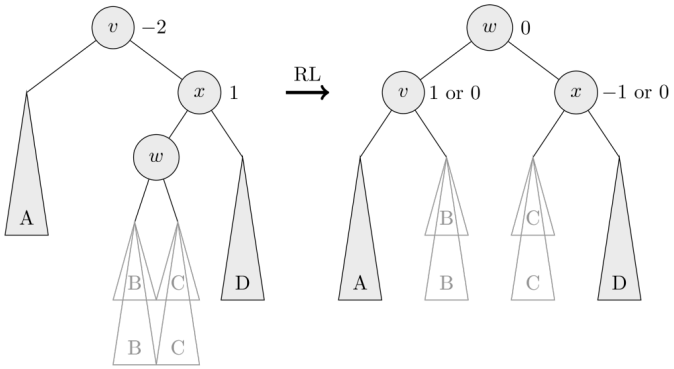
Case 1: $\text{bal}(v) = 1 \rightarrow \text{bal}(v) = 0$ **done.**

Case 2: $\text{bal}(v) = 0 \rightarrow \text{bal}(v) = -1$ **done.**

Case 3.1: $\text{bal}(v) \in \{-1, 0\}$ **and** $\text{bal}(v.right) \leq 0$



Case 3.2: $\text{bal}(v) \in \{-1, 0\}$ **and** $\text{bal}(v.right) > 0$



Lecture 11

- **Optimal substructure**: The solutions to the subproblems used within the optimal solution must themselves be optimal.
- **Bellman equation**: 状态转移方程
- **Bottom-up implementation**: 先解最小的 case，然后更大的 case 用小的答案推出
- **Top down Implementation with Memorization**: 记忆化搜索

DP with Solutions

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1:  Let r[0...n] and s[0...n] be new arrays
2:  r[0] = 0
3:  for j = 1 to n do
4:      q = -∞
5:      for i = 1 to j do
6:          if q < p[i] + r[j - i] then
7:              q = p[i] + r[j - i]
8:              s[j] = i
9:      r[j] = q
10: return r and s
```

Current best solution cuts at *i*
Store this information in *s*.

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]          // cut location for length n
4      n = n - s[n]        // length of the remainder of the rod
```

Lecture 12

Correctness of the greedy choice (activity-selection problem)

- Define S_k as the set of activities that start after finishing a_k.
- **Theorem 15.1:** Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then
  a_m is included in some maximum-size subset of mutually compatible activities of S_k.
- In other words: there is a $\mathbf{maximum\text{-}size\ set}$ that includes the activity with earliest finish time (greedy choice). When applying the greedy choice we are still on track for finding a maximum-size set of activities. Hence the greedy choice is always safe.

General scheme for correctness of greedy algorithms

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

Lecture 13

Store the Graph

For $G = (V, E)$:

Adjacency-list representation: $|V|$ lists, the sum of the list lengths is $|E|$.

Adjacency-matrix representation: one $|V| \times |V|$ matrix.

BFS

```
BFS(G, s)
 1: for each vertex u ∈ V \ {s} do
 2:       u.colour = WHITE
 3:       u.d = ∞
 4:       u.π = NIL
 5: s.colour = GRAY
 6: s.d = 0
 7: s.π = NIL
 8: Q = ∅
 9: ENQUEUE(Q, s)
10: while Q ≠ ∅ do
11:       u = DEQUEUE(Q)
12:       for each v ∈ Adj[u] do
13:             if v.colour = WHITE then
14:                   v.colour = GRAY
15:                   v.d = u.d + 1
16:                   v.π = u
17:                   ENQUEUE(Q, v)
18:       u.colour = BLACK
```

```
PRINT-PATH(G, s, v)
1  if v == s
2        print s
3  elseif v.π == NIL
4        print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6        print v
```

runtimes: $\Theta(|V| + |E|)$ for the whole graph.

Theorem: Let $G(V, E)$ be a graph, and BFS is run on source $s \in V$. Then BFS discovers every vertex $v \in V$ that is reachable from s, and upon termination $v.d = \delta(s, v)$ **for all** $v \in V$ (note: $\delta(u, v)$ means the shortest path from u to v).

Lecture 14

DFS

timestamps:

- $v.d$ is the time v is first **discovered** (and grayed)
- $v.f$ is the time v is **finished** (and blackened)
- Global variable time is incremented with each event. Hence for all vertices $v.d < v.f$

```
DFS(G)
1: for each vertex u ∈ V do
2:       u.colour = white
3:       u.π =NIL
4: time = 0
5: for each vertex u ∈ V do
6:       if u.colour == white then
7:             DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1: time = time+1
 2: u.d = time
 3: u.colour = gray
 4: for each v ∈ Adj[u] do
 5:       if v.colour == white then
 6:             v.π = u
 7:             DFS-VISIT(G, v)
 8: u.colour = black
 9: time = time+1
10: u.f = time
```

runtime: also $\Theta(|V| + |E|)$ for the whole graph.

Parenthesis structure: In any DFS of a (directed or undirected) graph, for any two vertices $u \neq v$, either

- $\text{DFS-Visit}(v)$ is called during $\text{DFS-Visit}(u)$, then v is a descendant of u and $\text{DFS-Visit}(v)$ finishes earlier than u: $u.d < v.d < v.f < u.f$

Theorem: In a depth-first forest of a (directed or undirected) graph, vertex v is a descendant of a vertex u **if and only if** at the time $u.d$ that the search discovers u, there is a path from u to v consisting entirely of white vertices.

Edges

1. **Tree edges** are edges in the depth-first forest. Edge (u, v) is a tree edge if, v was first discovered by exploring edge (u, v).
   An edge (u, v) is a tree edge if at the time of exploration v is white.
2. **Back edges** are edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree (or self-loops in directed graphs).
   An edge (u, v) is a back edge if at the time of exploration v is gray.
3. **Forward edges** are nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree (pointing forward in the tree).
   (u, v) is a forward edge if v is black and was discovered later: $u.d < v.d$.
4. **Cross edges** are all other edges: either leading to a subtree constructed earlier or leading to a different (earlier) depth-first tree.
   (u, v) is a cross edge if v is black and was discovered earlier: u.d > v.d.

Theorem: In a depth-first search of an undirected graph, every edge is either a **tree** edge or a **back** edge.

Theorem: A graph G contains a cycle if and only if DFS finds **at least one back edge**.

Topological Sorting

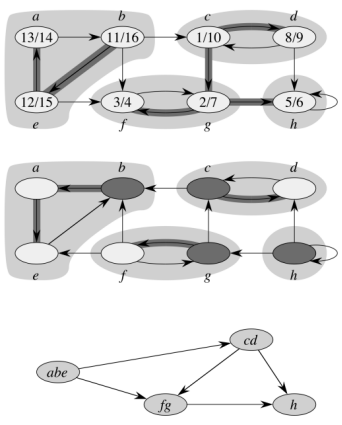
- condition: the graph is **directed and acyclic**.

TOPOLOGICAL-SORT(G)
1: call DFS(G) to compute finishing times $v.f$ for each vertex v
2: as each vertex is finished, insert it onto the front of a linked list
3: **return** the linked list of vertices

runtime: $\Theta(|V| + |E|)$.

Strongly Connected Components

STRONGLY-CONNECTED-COMPONENTS(G)
1: call DFS(G) to compute finishing times $v.f$ for each vertex v
2: compute $G^\top$
3: call DFS($G^\top$), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4: output the vertices of the tree in the depth-first forest formed in line 3 as a separate SCC



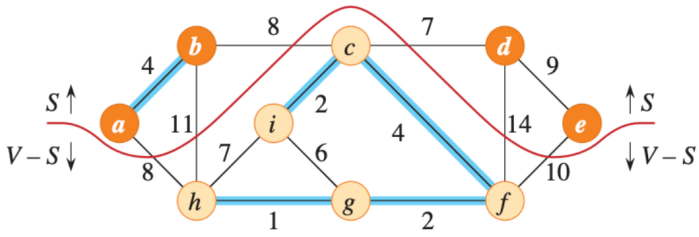
runtime: $\Theta(|V| + |E|)$.

Lecture 15

Minimum Spanning Tree

Cuts

- A cut of an undirected graph $G = (V, E)$ is a partition of V in two sets $(S, V \setminus S)$.



- An edge **crosses** the cut if exactly one of its endpoints is in S.
- A cut **respects** a set A of edges if no edge in A crosses the cut.
- An edge is a **light edge** if its weight is minimal among all edges with some property, e. g. for all edges crossing the cut.

Theorem 23.1: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some MST for G. **If $(S, V \setminus S)$ is a cut of G that respects A, and (u, v) is a light edge crossing $(S, V \setminus S)$, then (u, v) is safe for A.**

Prim

```
PRIM(G, w, r)
1: for each vertex u ∈ V do
2:     u.key = ∞
3:     u.π = NIL
4: r.key = 0
5: Q = V
6: while Q ≠ ∅ do
7:     u = EXTRACT-MIN(Q)
8:     for each v ∈ Adj[u] do
9:         if v ∈ Q and w(u, v) < v.key then
10:            v.π = u
11:            v.key = w(u, v)
```

```
PRIM(G, w, r)
1: for each vertex u ∈ V do
2:     u.key = ∞
3:     u.π = NIL
4: r.key = 0
5: Q = V
6: BUILD-MIN-HEAP(Q)
7: while Q ≠ ∅ do
8:     u = EXTRACT-MIN(Q)
9:     for each v ∈ Adj[u] do
10:        if v ∈ Q and w(u, v) < v.key then
11:            v.π = u
12:            DECREASE-KEY(Q, v.key, w(u, v))
```

runtime: $\Theta(|V|^2)$ for basic algorithm; use heap we and adjacency-list we can do it in $O(|E| \log |V|)$.

Kruskal

```
KRUSKAL(G, w)
1: A = ∅
2: for each vertex v ∈ V do
3:     make a set {v}
4: sort the edges of E in nondecreasing order by weight w
5: for each edge (u, v) in this order do
6:     if FIND-SET(u) ≠ FIND-SET(v) then
7:         A = A ∪ {(u, v)}
8:         UNION(u, v)
9: return A
```

Shortest Path (Dijkstra)

```
DIJKSTRA(G, w, s)
1: Initialise d and π in the usual way.
2: S = ∅
3: Q = V
4: while Q ≠ ∅ do
5:     u = EXTRACT-MIN(Q)
6:     S = S ∪ {u}
7:     for each v ∈ Adj[u] do
8:         if v.d > u.d + w(u, v) then
9:             v.d = u.d + w(u, v)
10:            v.π = u
11:            DECREASE-KEY(Q, v, v.d)
```

Note: Dijkstra cannot solve the problem that **the weight of some of edges are negative**.