Designing Algorithms using Induction and Recursion

Acknowledgement
This notes is compiled based on the following textbooks and a variety of Internet resources, such as Wikipedia for example.

- Introduction to Algorithms by Cormen, Lerserson, Rivest and Stein.
- Algorithms by Dasgupta, Papadimitriou and Vazirani
- Introduction to Algorithms, A Creative Approach by Manber.

1 Introduction

Mathematical induction is a method of proof to establish the correctness of an argument $P(n)$ with respect to natural numbers $n = 1, 2, 3, ...$ An induction proof has two parts, the first part is called the **basis** that establish the correctness of of the arguent $P(n)$ for $n = 1$. The second part is called the **induction step**, which starts with an **induction hypothesis** that assumes the argument $P(k)$ is correct for any natural number k, and then show that if $P(k)$ is true than $P(k + 1)$ is true. The **strong mathematical induction** proof strengthens the hypothesis, by assuming that the claim is true all natural numbers $1, 2, ..., k$ and use the hypotheis to show that $P(k + 1)$ is true.

The earliest example of mathemtical induction or inductive argument can be found Plato's (427-347 BC) Parmenides, which describes a conversation between Parmenides (515-460BC) and Socrates (470/469-399BC). (Note that according to http://classics.mit.edu/Plato/parmenides.html, the dialogue was between Parmenides and Aristotle (384-322BC), however, this must be a typo.) Figure 1 shows the portraits of these four great Greek Philosophers, and here is the dialogue:

> Two things, then, at the least are necessary to make contact possible?
>
> They are.
>
> And if to the two a third be added in due order, the number of terms will be three, and the contacts two?
>
> Yes.
>
> And every additional term makes one additional contact, whence it follows that the contacts are one less in number than the terms; the first two terms exceeded the number of contacts by one, and the whole number of terms exceeds the whole number of contacts by one in like manner; and for every one which is afterwards added to the number of terms, one contact is added to the contacts.
>
> True.

Whatever is the whole number of things, the contacts will be always one less.

True.

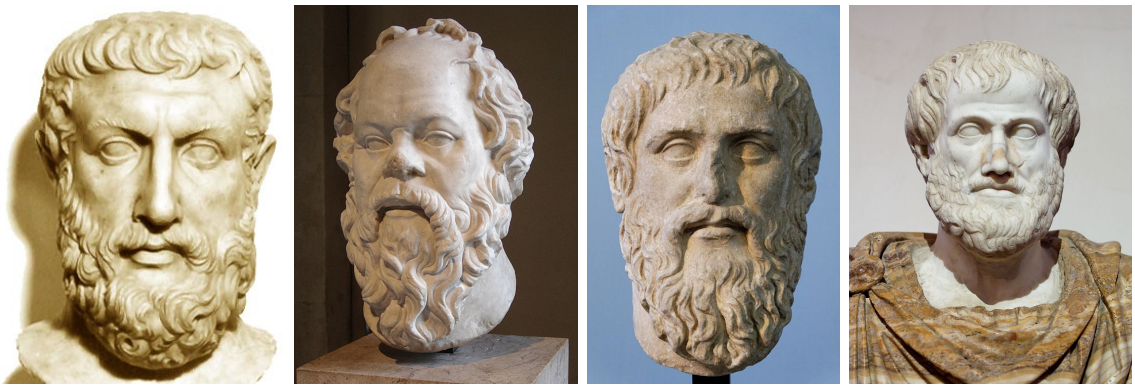

Figure 1: Some of the greates Greek Philosophers: Parmenides, Socrates, Plato and Aristotle.

We now discuss how to use the inductive and recursive strategies for developing algorithms. The inductive strategy uses the following key steps:

1. Solve a small instance of the program.
2. Assuming that we can solve all possible instances of the problem of size $< n$, develop the solution to the problem of input size n by assuming that the solutions to problem sizes of $< n$ are readily available.

the recursive or **divide-and-conqure** strategy uses the following key steps:

1. Breaking down a problem into a collection of smaller sub-problems, which are easier to solve.
2. Recursively solve each of the sub-problems.
3. Combining the solutions for each sub-problem to form the solution to the original problem

Observe that the two strategies are closely related. Infact, the induction step of the inducation strategy is essentially the dividing and recursion step of the divide and conquer strategy.

2 Insertion Sort

Problem 1 *Given an array A of n real numbers, sort the numbers.*

The insertion sort algorithm is a good and simple example for us to illustrate the idea of using indution to design algorithms.

Basis: Clearly if we are only given $n = 1$ number, it is already sorted.

I.H. Now assume that we know how to sort any given $n - 1$ numbers.

I.S. Now assume that we are given n numbers. With the induction hypothesis, we can assume that the first $n - 1$ numbers can be sorted. The key question here is how can we incorporate the $n - th$ number? Since the first $n - 1$ numbers are already sorted, all we need to do is to **insert** the $n - th$ number into a list of the $n - 1$ sorted numbers, which is very easy to do by simply scanning the sorted $n - 1$ numbers and fin the place to insert the $n - th$ number.

This immediately yields the following insertion sort algorithm:

Algorithm: Insertion Sort

Input: An array A of n real numbers.

Output: The array A sorted in non-decreasing order.

1. for $j = 2$ to n:
2. for $k = j$ to 2
3. if $A[k] < A[j]$, swap $A[k]$ and $A[j]$;
4. else break;

The running time of the above algorithm in the worst case is

$$\sum_{j=2}^{n}\sum_{k=j}^{2} = \sum_{j=2}^{n}(j-1) = 1 + 2 + ... + (n-1) = \frac{(n-1)n}{2} = \Theta(n^2)$$

3 Merge Sort

Problem 2 *Given an array A of n real numbers, sort the numbers.*

Now let's use divide and conquer to sort the array A.

Divide: We will partition the array $A[1..n]$ into two sub-arrays $A[1..\frac{n}{2}]$ and $A[\frac{n}{2}+1, ..., n]$, Exit condition: we stop dividing the array when the array only has 1 element.

Recursion: Recursively sort both arrays.

Conquer: Combine/merge the two sorted sub-arrays return from the recursions into a single sorted array.

To merge two already sorted arrays, we will use the following idea. Suppose that we are given two input piles of cards facing up on a table. Each pile is already sorted, and the smallest cards are on top. We need to combine these two piles into one output pile. The output file must also be sorted and facing down on the table, where the smallest cards are at the bottom. To achieve this, we will need to repeatedly remove the smallest card from the two input piles and place it face down on the output pile. Since both input piles are already sorted, all we need to do is to compare their top cards and remove the smaller of the two. The input pile with the top card removed to the output pile will then expose its new current smallest card. The process is repeated until all the input piles are exhausted. If during this process, any one of the pile is empty, we shall assume its current smallest card is ∞.

The pseudo-code for the merge-sort algorithm are as follows:

Algorithm:

Input: An array A of n real numbers.

Output: The array A sorted in non-decreasing order.

1. Merge-Sort (A, 1, n) ;

Algorithm: Merge-Sort

Input: An array A, a starting index s and an ending index t.

Output: $A[s..t]$ sorted

1. if $s = t$, return;
2. $r = \lfloor \frac{t-s+1}{2} \rfloor$
3. Merge-Sort (A, s, r) ;
4. Merge-Sort (A, r+1, t) ;
5. Merge (A, s, r, t) ;

Algorithm: Merge

Input: An array A and three indices s, r, t, where $A[s..r]$ and $A[r+1..t]$ are both sorted.

Output: Combine $A[s..r]$ and $A[r+1..t]$ into $A[s..t]$ sorted

1. introduce a working array B of size $t-s+1$.
2. $i = s$, which indexes the top card of the first input pile
3. $j = r+1$, which indexes the top card of the second input pile
4. $k = 1$, which indexes the top card of the output array B.
5. while $i \leq r$ and $j \leq t$
6. if $A[i] \leq A[j]$, $B[k] = A[i]$, $i++$, $k++$
7. else $B[k] = A[j]$, $B[k] = A[j]$, $j++$, $k++$
8. if $i > r$, the pile $A[s, r]$ is empty, copy everything from $A[j..t]$ to B.
9. else the pile $A[r+1..t]$ is empty, copy everything from $A[i..r]$ to B.

The running time of the above algorithm in the worst case is

$$T(n) = 2T(\frac{n}{2}) + n = \Theta(n \log n).$$

4 Radix Sort

Problem 3 *Suppose you are given an array A of n non-negative integers. Each integer in A has at most k digits, which means the range of the integers in A are from 0 to $10^k - 1$. Sort A in non-decreasing order.*

We use the induction on k, the number of digits to develop the Radix Sort algorithm.

Basis: If $k = 1$, the numbers in A have only 1 digit, meaning the range of the n numbers are from 0 to 9. For this, we will create 10 buckets. Each bucket is a queue data structure. (Recall that a queue is a first-in-first-out data structure, while a stack is a last-in-first-out data structure.) We will denote the buckets as $Q_0, Q_1, Q_2, ..., Q_9$. For each number $A[j]$, will put put into bucket $Q_{A[j]}$. We can then collect the contents of all the buckets in the increasing order of the bucket index and put them back into A. This will gives a sorted array A. Since each element in A will only be access twice, one time for putting in a bucket, and the other time for retrieving it from the bucket, the total running time is linear.

Note that the algorithm described in the Basis is often referred to as **bucket sort**.

I.H. Assume that we know how to sort if the given numbers have $< k$ digits.

I.S. Assume that we are given n numbers with k digits. With the induction hypothesis, we may assume that all the numbers in A are already sorted based on the least $k - 1$ significant digits. What if we run another bucket sort based on the most significant digit?

Consider two numbers $a = (a_{k-1}a_{k-2}...a_0)$ and $b = (b_{k-1}b_{k-2}...b_0)$ from A, where a_{k-1} and b_{k-1} denote the most signficant digit. If $a_{k-1} < b_{k-1}$, then obviously, a must be in front of b in the final sorted array. This is obvously the case after the bucket sort with most significant digit. Similarly, if $a_{k-1} > b_{k-1}$, a must be after b, which is also guaranteed after the final round of bucket sort. Finally, if $a_{k-1} = b_{k-1}$, the order of a and b will be determined by their least $k - 1$ significant digits. Without loss generality, assume that $(a_{k-2}...a_0) < (b_{k-2}...b_0)$. Now from the induction hypothesis, we know that a will be before b before we started the bucket sort using the most significant digit. They will be placed in the same bucket in the bucket sort based on the most significant digit. a is in front of b before the bucket sort, and each bucket is a queue, a will still be befor b after the bucket sort. Thus the correct order between a and b is preserved after the bucket sort based on the most significant digit.

The pseodo-code of the above algorithm are as follows:

Algorithm: Radix Sort

Input: An array A of n non-negative $k-$digit integers

Output: A sorted

1. Create an array B of 10 buckets, each bucket is a queue.
2. for $j = 0$ to $k - 1$

3. for $i = 1$ to n

4. Put $A[i] = (d_{k-1}d_{k-2}...d_0)$ into $B[d_j]$.

5. $i = 1$

6. for $l = 0$ to 9

7. while bucket B[i] is not empty

8. $A[i] = dequeue(B[l])$, i ++ ;

Running time: $O(nk)$. Note that if k is a constant or much smaller than $\log n$, then the radix sort considerably outperforms merge sort.

5 Sorting using a Heap and a BST

Definition 1 *A* **graph** $G(V, E)$ *consists of a set* V *of* **vertices** *(or* **nodes***) and set* E *of* **edges***. Each edge connects a pair of vertices.*

A graph is a **directed** graph (or **digraph**) if its edges are ordered pairs. For a directed edge $e = (v, w)$, we called v the **head** of the edge and w the **tail**. In a digraph, the **indegree** of a vertex v is the number of edges whose tails are v, and **outdegree** is the number of edges whose heads are v.

A graph is **undirected** if its edges are unordered pairs. For an undirected edge e connecting two vertices v and w, we say the edge e is **incident** to v and w. In an undirected graph, the **degree** of a vertex v is the number of edges incident to it.

An edge $e \in E$ that connects the same vertices is called a **self loop**. Unless specified otherwise, we shall assume that self loops are forbidden.

A **multigraph** is a graph with possibly multiple edges between the same pair of vertices. A graph is said to be **simple** is it has no self loops nor multiple edges between the same pair of vertices. Unless specified otherwise, we will assume that the graphs we deal with are simple.

For an undirected graph, a **path** from a vertex v_0 to v_k is a sequence of vertices and edges v_0, e_1, v_2, e_2, v_3, ..., v_k, where the edge e_j connects v_{j-1} and v_j. For a digraph, a **path** or **directed path** from a vertex v_0 to v_k is also a sequence of vertices and edges v_0, e_1, v_2, e_2, v_3, ..., v_k, where each edge e_j is the ordered pair (v_{j-1}, v_j).

A vertex w is said to be **reachable** from a vertex v if there is a path from v to w. A graph is said to be **connected** if there is a path between any pair of distinct vertices.

A path is a **simple path** if each vertex appears at most once. A **circuit** is a path whose first and last vertics are the same. A circuit is **simple** if except the first and last vertices, no vertex appears more than once. A simple circuit is also called a **cycle**.

A **subgraph** of a graph $G(V, E)$ is a graph $H(U, F)$ such that $U \subseteq V$ and $F \subseteq E$.

Let $G(V, E)$ be a graph and $U \subseteq V$, then an **induced subgraph** of G by U is a subgraph $H(U, F)$ of G such that F consists of all edges in E whose both vertices belong to U.

Definition 2 *A* **forrest** *is a undirected simple graph with no cycles.*

Definition 3 *A* **tree** *is a connected undirected simple graph with no cycles.*

There are some well-known properties of trees.

- There is a unique simple path between any two pair of vertices.
- A tree with n vertices has $n - 1$ edges.

Note that, you should be able to prove these properties.

Definition 4 *A* **rooted tree** *is a tree with one node designated as the root.*

Since there is a unique simple path between any pair of vertices in a tree, there is a unique path from the root v to any other vertex v. Let this path be $v_0 = r$, v_1, v_2, ..., v_{k-1} $v_k = v$. The node v_{k-1} is called the **parent** of v, and v is called a **child** of v_{k-1}. The vertices $v_0 = r$, v_1, ..., v_{k-1} are called

the **ancestors** of v, and v is called the **descendents** of $v_0 = r$, v_1, ..., v_{k-1}. The number k is the **level** of v. (The root has level 0.) Since the level of every node is well-defied, a rooted tree defines a hierachy, i.e., level 0 (i.e., the root), level 1, level 2, ...

For a node v in a tree, the **subtree** rooted at v is the subgraph of the tree induced by the vertex v and all its descendents.

The maximum number of children of any node in the tree is called the **degree** of the tree.

A tree of degree 2 is also called a **binary tree**. Since every node in a binary tree can have at most two children, we shall denote one such child as the **left child** and the other one (if available) as the **right child**. The sub-tree rooted at the left child is called the **left sub-tree** and the sub-tree rooted at the right child is called the **right sub-tree**.

The nodes in a tree may have a data field containing data that is associated with the node. Among many things that the data field may contain is a **key** from a totally ordered set.

5.1 Sorting using a Heap

Definition 5 *A* **heap** *or* **priority queue** *is a logically binary tree, that is completely filled on all levels except possibly the largest, which is filled from the left up to a point.*

Observation 1 *The height of a heap with n nodes is $\Theta(\log n)$.*

Proof: Let the height be h. Since the underlying binary tree is complete, the number of nodes in the heap is bounded by $\sum_{j=0}^{h-1} 2^j + 1 \leq n \leq \sum_{j=0}^{h} 2^j$. Thus, $2^h \leq n \leq 2^{h+1} - 1$. Hence $\log(n+1) - 1 \leq h \leq logn$. □

Definition 6 *A* **max-heap** *is a heap where the key of every node is greater than or equal to the key of any of its children.*

Definition 7 *A* **min-heap** *is a heap where the key of every node is less than or equal to the key of any of its children.*

Physically, a heap is usually stored in an array, where nodes of the logical binary tree underneath the heap are placed in the array starting from the root, and level by level, and from left to right within each level. A number called heapsize indicating the number of the nodes in the heap is also maintained. The parent-child relation is implicit because the underlying binary tree is complete. Assuming that the array index starts with 1, then for a node indexed j, its parent is $\lfloor \frac{j}{2} \rfloor$, the left child of j is $2j$ and the right child is $2j + 1$.

The key for using a heap is to maintain the the heap property (i.e., max-heap or min-heap property). The heap data structure supports 3 key operations: heapify, extract the key and remove the root, and heap-insert. Below, we shall use a max-heap to illustrate these three operations:

1. max-heapify: The input is a node index j in a max-heap H. Heapify assumes that the left and right sub-trees of j are all max-heaps, but due to certain reasons j may be violating the heap property. The key idea of max-heapify is to let the node j "float down" so that the subtree rooted j is a max-heap.

2. extract-max: The largest key of a heap is stored in its root. Extract-max aims to report the key of the root, and delete the root of the heap. To ensure the remaining heap is still a valid max-heap, extract-max will take the last element of the heap array and place it into the root, and then invoke heapify to update the heap.

3. max-heap-insert: The goal is to insert a key into the heap. The key idea of heap-insert is to place the new key at the end of the heap, and then let the new key "float up" to ensure the new heap is a valid max-heap.

Algorithm: max-heapify

Input: A node j in a max-heap H.

Output: Ensures that the sub-tree rooted at j is a max-heap

1. Let the l be the index of the child of j with the larger key.

2. if $H[j] < H[l]$, then swap the keys of j and l, and max-heapify (H, l)

The running time for heapify is height of the sub-tree rooted at j.

Algorithm: extract-max

Input: A max-heap H

Output: Removing the root of H, return the key of the root, and update H to maintain max-heap property.

1. $root\_key = H[1]$.
2. $H[1] = H[heap\_size]$
3. $heap\_size - -$
4. max-heapify $(H, 1)$.
5. return $root\_key$

The running time for extract-max is the heap of the heap H, and therefore $O(\log n)$.

Algorithm: max-heap-insert

Input: A max-heap H and a key x to be inserted.

Output: An updated heap H with x.

1. $heap\_size + +$.
2. $H[heap\_size] = x$
3. $j = heap_size$
4. while $j > 1$ and $h[parent\ of\ j] < H[j]$
5. Swap j and its parent
6. Set j to its parent

The running time for extract-max is the heap of the heap H, and therefore $O(\log n)$.

Generally speaking, heap sort algorithm has two steps: (1) Build a max heap for all the numbers to be sorted, and (2) While the heap is not empty, repeated involing extract-max.

Algorithm: Heap Sort

Input: An array A of n real numbers.

Output: The numbers in A sorted in non-increasing order.

1. Create a heap H of size n, $heap_size = 0$.

2. For $j = 1$ to n:

3. $\quad heap - insert(H, A[j])$

4. For $j = 1$ to n:

5. $\quad A[j] = extract - max(H)$

The above heap sort algorithm takes $O(n \log n$ time. The problem is the use of linear additional space. One could save the space by using heapify to convert the given array A into a max-heap.

Algorithm: Heap Sort with No Additional Space

Input: An array A of n real numbers.

Output: The numbers in A sorted in non-decreasing order.

1. For $j = \lfloor n \rfloor$ down to 1

2. $\quad heapify(A, j)$.

3. For $j = n$ to 1:

4. $\quad$ Swap $A[j]$ with $A[1]$

5. $\quad heap\_size = j$

6. $\quad heapify(A, 1)$

The running time of this second version of the heap sort algorithm has two component. The second component is obviously $O(n \log n)$. The more interesting part is the analysis of the running time of the first part for building the heap.

The time for building the heap is

$$\sum_{h=0}^{\log n} \frac{n}{2^h} h = n \sum_{h=0}^{\log n} \frac{h}{2^h} \leq n \sum_{h=0}^{\infty} \frac{h}{2^h}$$

Let $S = \sum_{h=0}^{\infty} \frac{h}{2^h}$ Observe that $\frac{S}{2} = \sum_{h=0}^{\infty} \frac{h}{2^{h+1}}$. Thus

$$S - \frac{S}{2} = \sum_{h=0}^{\infty} \frac{h}{2^h} - \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} = \sum_{h=1}^{\infty} \frac{h}{2^h} = 1$$

Thus $S = 2$, and the time for building the heap is $O(n)$.

5.2 Sorting using a Binary Search Tree

Definition 8 *A* **binary search tree** *or in short* **BST** *is a binary tree, such that the key of every node is greater than all the keys in its left sub-tree and less than all the keys in its right sub-tree.*

The key operations supported by a BST is search, insert and delete.

Algorithm: Search

Input: A BST T and a key x.

Output: If x is in T, return the node containing x, or nil otherwise

1. Let r be the root of the T.
2. while $r \neq nil$ do:
3. if $x = r.key$, return r
4. else if $x < r.key$, $r = r.left$.
5. else $r = r.right$.
6. return nil

Algorithm: Insert

Input: A BST T and a key x to be inserted

Output: An updated T with x inserted.

1. Let r be the root of the T.
2. if $r = nil$, we have an empty tree, and let's place x into the root.
3. while $r \neq nil$ do:
4. if $x = r.key$, the key is already in the tree.
5. else if $x < r.key$, $r = r.left$.
6. else $r = r.right$.
7. $r = r.parent$
8. if $x < r.key$, $r.left = x$, else $r.right = x$.

Algorithm: Delete

Input: A BST T and a key x to be deleted

Output: An updated T with x inserted.

1. $z = search(T, x)$.

2. if $z = nil$, return.

3. else

4. if z is a leaf, we can simply remove it.

5. else we need to find the largest key $< z.key$, i.e., x or the smallest key $> z.key$ to replace z.

Observe that the running time for search, insert and delete of a BST are all bounded by the height of the tree.

Unlike heap/priority queue, which is always balanced (i.e., the differene between the height of left and right sub-trees are no long a constant), the BST may be unblanced after insertion and deletion, causing the height of the tree to be $\omega(\log n)$, and therefore causing poor performance. To overcome this, a number of balanced binary search trees have been developed, such as the AVL tree and red-black tree, which guarantees $O(\log n)$ time for searching, insertion, and deletion.

Assuming that we are using a balanced binary search tree, then one can use a BST for sorting as well. Like the heap sort, the algorithm also has 2 phases. In the firt phase, a balanced binary search tree is built, and in the second phase, the in-order traversal is performed on the BST and the output keys are sorted.

6 AVL Tree, an Example of a Balanced BST

A BST is said to be "balanced" if it can achieve searching, insertion and deletion in $O(\log n)$ time. In this lecture, we will look at the AVL tree, which is an example of a balanced BST.

Definition 9 *The AVL-tree, named after its inventors Adelson-Velskii and Landis, is defined to be a BST such that for every node, the height difference between its left and right sub-trees is at most 1. (See Figure 2 for illustrations.)*

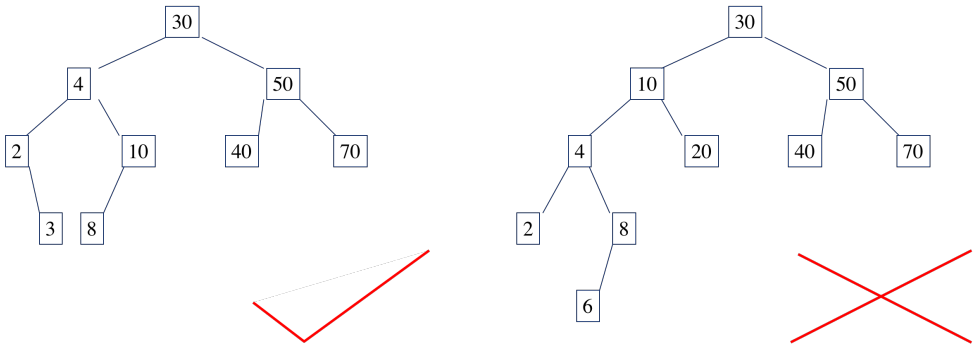

Figure 2: Illustration of an AVL tree.

Observation 2 *The height of an AVL-tree with n nodes is bounded by $2\log n$. In other words, for an AVL-tree of height h, the number of nodes in the tree n is $\geq 2^{\frac{h}{2}}$*

Proof: We will prove this by induction on the height h of the AVL-tree.

Basis: $h = 0$. This has to be a singleton tree with $n = 1$. Thus $n \geq 2^{\frac{h}{2}} = 2^0 = 1$.

Induction Hypothesis: Assume that for all AVL-trees with height $\leq h$, the claim is correct

Induction Step: Consider an AVL-tree of height $h+1$. Consider the two sub-trees of the root. We have two possibilities: (1) both sub-trees have a height h, or (2) one of them has a height $h-1$ and the other had a height h.

In case (1), we have $n \geq 1 + 2 \times 2^{\frac{h}{2}} = 1 + 2^{\frac{h}{2}+1} \geq 2^{\frac{h+1}{2}}$.

In case (2), we have $n \geq 1 + 2^{\frac{h}{2}} + 2^{\frac{h-1}{2}} \geq 1 + 2 \times 2^{\frac{h-1}{2}} \geq 2^{\frac{h-1}{2}+1} = 2^{\frac{h+1}{2}}$. □

Note that, one can tighten this bound by proving that the height of an AVL-tree is bounded by $1.4404\log(n+2) - 0.328$.

Since an AVL-tree is also a BST, searching an AVL-tree is the same as searching a BST.

Insertion and deletion in an AVL-tree is carried out in the following manner: (1) A normal BST insertion and deletion is performed on the AVL-tree. (2) If after the normal insertion and deletion, the AVL-properties are violated, a single or double rotation is performed on the tree to restore the AVL-properties. Both the single and double rotations are cheap and can be performed in constant time. In the case of insertion, only one rotation is needed, while in the case of deletion, at most $O(\log n)$ rotations are needed. Therefore the running time for insertion and deletion in an AVL-tree is bounded by $O(\log n)$ time.

6.1 Insertion in an AVL-tree

Let A be the root of the smallest subtree of an AVL-tree that violates the AVL property, i.e., the height difference between the two subtrees of A is 2 after performing a normal insertion.

Let B be the leftchild of A and C be the rightchild of A. Without loss of generality, we shall assume that after the normal insertion, the height of the subtree rooted at B is more than 2 taller than the height of the sub-tree rooted at C.

For ease of explanation, let's assume before the insertion, the subtree rooted at A has a height $h+1$, the subtree rooted at B has a height h, the subtree rooted at C has a height $h-1$.

Further, let D be the leftchild of B and E be the rightchild of B. The height of the subtrees rooted at D and E are both $h-1$. Note that if these two subtrees have a height difference, then B instead of A should be the smallest subtree violating the AVL property.

Since the new key may either be inserted in the subtree of D or E, we have two scenarios to consider here.

Let's first consider when the insertion happens to the subtree rooted at D. Note that this increases the height of the subtree rooted at D to h, the subtree rooted at B to $h+1$, which now has a height difference of 2 with the subtree rooted at C, making A the smallest subtree violating the AVL property. Our goal is to make a local change to reduce the height of the subtree rooted at A to its original height, so that we don't need to worry about the rest of the AVL tree. The solution is to "elavate" B to replace A as the new root of the subtree rooted at A. We will then attach E to be the new leftchild of A. (See Figure 3 for illustrations.)

Now let's consider the other situations, i.e., when the insertion happens to the subtree rooted at E. Let F and G be the left and right child of E respectively. Then the subtrees rooted at F and G has a height $h-2$ Without loss of generality, we shall assume that the insertion happens to the subtree rooted at F, which increases the subtree rooted at F to $h-1$, and the subtree rooted at E to h, and the subtree rooted at B to $h+1$. In this situation, we shall "elevate" E to be new root to replace A. We will then attach B to E as the left child and A to E as the right child. We will then attach D to B as the left child, and F to B as the right child. We will also attached G to A as the left child. Thus restoring the height of the subtree to the original $h+1$. (See Figure 4 for illustrations.)

6.2 Deletion in an AVL Tree

The deletion is the AVL tree is very similar to the insertion. We perform a normal deletion just like in a regular BST. If deletion causes the tree to be no longer AVL, we will perform single or double rotations to rebalance the tree. The only caveat is that since the deletion may reduce the height or a sub-tree, we may have to perform as many as $O(\log n)$ deletions. (See Figure 5 for illustrations of deletions in an AVL tree.)

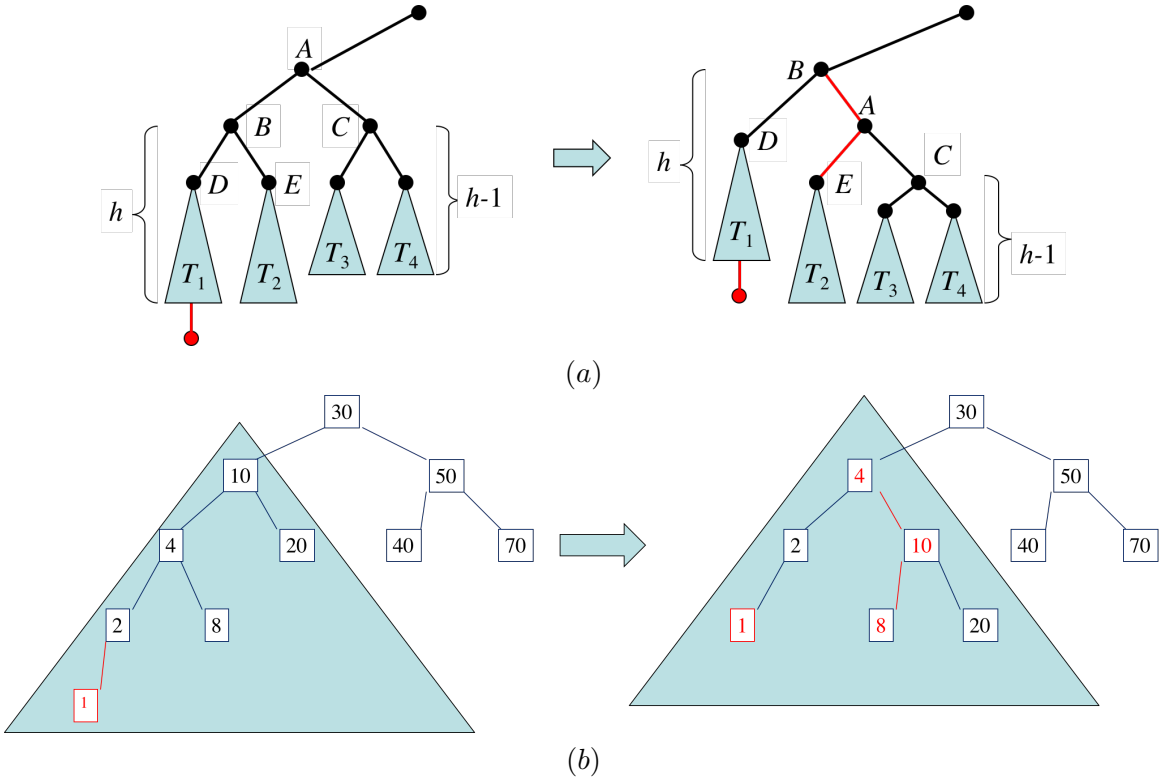

Figure 3: (a) Illustration of the single rotation in AVL tree insertion. (b) An detailed example.

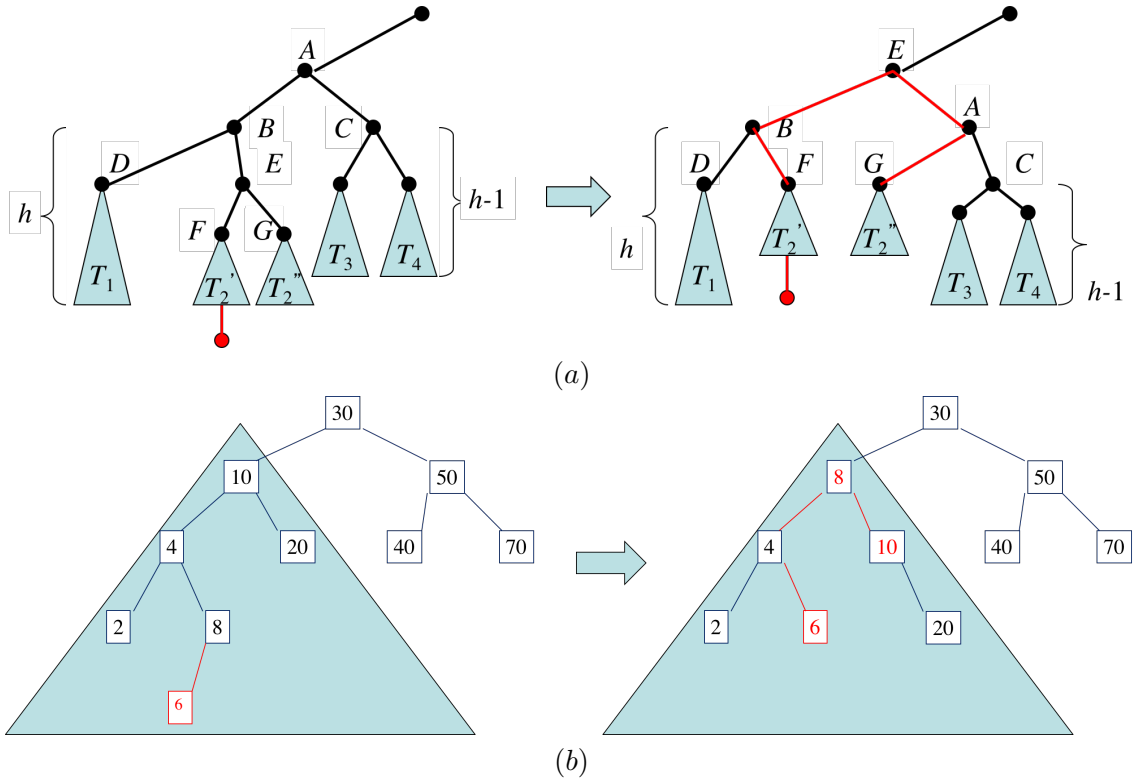

Figure 4: (a) Illustration of the double rotation in AVL tree insertion. (b) An detailed example.

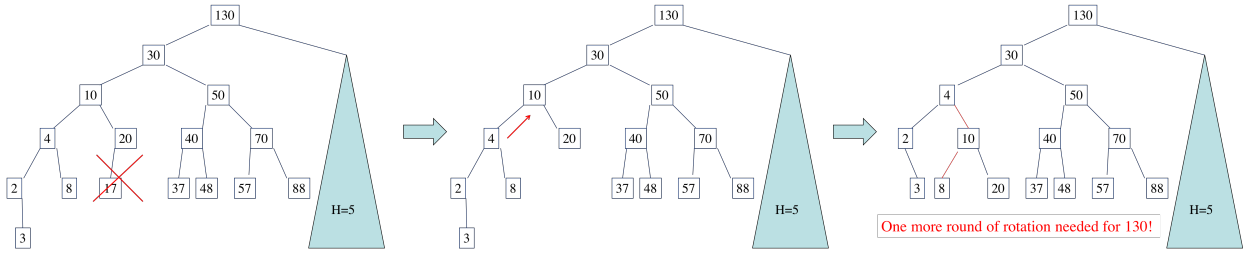

Figure 5: Illustration of the deletion in an AVL tree.

7 Polynomial Multiplication

Problem 4 *Given two $n-$th degree polynomials $P(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$ and $Q(x) = b_0 + b_1x + b_2x^2 + ...b_nx^n$, design an efficient algorithms for multiplying them.*

Let $R(X) = P(x)Q(x) = c_0 + c_1x + c_2x^2 + ... + c_{2n}x^{2n}$. Since for $0 \leq j \leq n$, $c_j = \sum_{k=0}^{j} a_k b_{j-k}$, and for $n+1 \leq j \leq 2n$, $c_j = \sum_{k=j-n}^{n} a_k b_{j-k}$, a straightforward way to calculate $R(x)$ will take $\sum_{j=0}^{n}(j+1)+\sum_{j=n+1}^{2n}(n-(j-n)+1) = \sum_{j=0}^{n}(j+1)+\sum_{j=n+1}^{2n}(2n-j+1) = \frac{(n+1)(n+2)}{2}+\frac{n(n+1)}{2} = (n+1)^2$ multiplications.

The interesting question here is can we reduce the number of multiplcations? It turns out that there is a divide and conquer algorithm that can reduce the number of multiplications signficantly.

The algorithm starts with an important observation by Gauss (1777-1855) that the product of two complex numbers $(a+bi)(c+di) = (ac-bd)+(bc+ad)i$ can be obtained using only three multiplications, i.e., $(a+bi)(c+di) = (ac-bd)+((a+b)(c+d)-ac-bd)i$, with the expense of 4 additions. Similarly, $(a_0+a_1x)(b_0+b_1x) = a_0b_0+((a_0+a_1)(b_0+b_1)-a_0b_0-a_1b_1)x+a_1b_1x^2$ can also be calculated using only three multiplications with the expense of 4 additions.

Consider partitioning the polynomial $P(x)$ as:

$$\begin{aligned} P(x) &= a_0 + a_1x + a_2x^2 + ... + a_nx^n \\ &= \left(a_0 + + ... + a_{\frac{n}{2}}x^{\frac{n}{2}}\right) + \left(a_{\frac{n}{2}+1}x^{\frac{n}{2}+1} + ... + a_nx^n\right) \\ &= \left(a_0 + + ... + a_{\frac{n}{2}}x^{\frac{n}{2}}\right) + x^{\frac{n}{2}}\left(a_{\frac{n}{2}+1} + a_{\frac{n}{2}+2}x + ... + a_nx^{\frac{n}{2}}\right) \\ &= P_0(x) + P_1(x)x^{\frac{n}{2}} \end{aligned}$$

Similarly,

$$Q(x) = Q_0(x) + Q_1(x)x^{\frac{n}{2}}$$

Now observe that:

$$P(x)Q(x) = \left(P_0(x) + P_1(x)x^{\frac{n}{2}}\right)\left(Q_0(x) + Q_1(x)x^{\frac{n}{2}}\right)$$

$$P_0(x)Q_0(x) + \left(P_0(x)Q_1(x) + P_1(x)Q_0(x)\right)x^{\frac{n}{2}} + P_1(x)Q_1(x)x^n$$

$$P_0(x)Q_0(x) + \left(\left(P_0(x) + P_1(x)\right)\left(Q_0(x) + Q_1(x)\right) - P_0(x)Q_0(x) - P_1(x)Q_1(x)\right)x^{\frac{n}{2}} + P_1(x)Q_1(x)x^n$$

This implies that we only need 3 recursions instead of 4. So, what's the impact of this on running time?

Let $T(n)$ be the running time of multiplying two $n-$degree polynomials. With the dividie and conquer strategy, we have $T(n) = 3T\left(\frac{n}{2}\right) + n$. Assume that $n = 2^k$, we have

$$\begin{aligned} & T(n) \\ &= 3T\left(\frac{n}{2}\right) + n \\ &= 3T\left(2^{k-1}\right) + 2^k \\ &= 3\left(3T\left(2^{k-2}\right)\right) + 2^{k-1} + 2^k \end{aligned}$$

$$
\begin{aligned}
&= 3^2 T\left(2^{k-2}\right) + 2^{k-1} + 2^k \\
&= ... = 3^k T(1) + 2^1 + 2^2 + ... + 2^{k-1} + 2^k \\
&= 3^k + 2^1 + 2^2 + ... + 2^k \\
&= \left(2^{\log_2 3}\right)^k + \left(2^{k+1} - 2\right) \\
&= \left(2^k\right)^{\log_2 3} + 22^k - 2 \\
&= n^{\log_2 3} + 2n - 2 \\
&= \Theta\left(n^{\log_2 3}\right) \approx \Theta\left(n^{1.585}\right)
\end{aligned}
$$

On the other hand, if we don't use the trick, we will end up with the recursion $T(n) = 4T\left(\frac{n}{2}\right) + n$. Assume that $n = 2^k$, we have

$$
\begin{aligned}
&T(n) \\
&= 4T\left(\frac{n}{2}\right) + n \\
&= 4T\left(2^{k-1}\right) + 2^k \\
&= 4\left(4T\left(2^{k-2}\right)\right) + 2^{k-1} + 2^k \\
&= 4^2 T\left(2^{k-2}\right) + 2^{k-1} + 2^k \\
&= ... = 4^k T(1) + 2^1 + 2^2 + ... + 2^{k-1} + 2^k \\
&= 4^k + 2^1 + 2^2 + ... + 2^k \\
&= \left(2^2\right)^k + \left(2^{k+1} - 2\right) \\
&= \left(2^k\right)^2 + 22^k - 2 \\
&= n^2 + 2n - 2 \\
&= \Theta\left(n^2\right)
\end{aligned}
$$

8 Selection Algorithm

Problem 5 *Given an array A of n real numbers, and an integer $1 \leq k \leq n$. Find the $k-th$ smallest number in A.*

Algorithm: Select

Input: An array A of n real numbers, and a positive integer $k \leq n$

Output: The $k-$th smallest number of A.

1. Partition the numbers of A into $\frac{n}{5}$ groups, each group has 5 numbers.
2. Find the median of each group and put these medians in an arrya M.
3. Recursively find the median of M, i.e., $x = Select(M, \frac{n}{10})$.
4. Use x as the pivot and partition A into two subsets $A_1 \leq x < A_2$.
5. If $|A_1| \geq k$, return $Select(A_1, k)$, ele return $Select(A_2, k - |A_1|)$

The running time of the algroithm is

$$T(n) = n + T(\frac{n}{5}) + T(\max\{|A_1|, |A_2|\}) = n + T(\frac{n}{5}) + T(\frac{n}{10}) = \Theta(n)$$

9 Dynamic Programming

Problem 6 *Let $S = \{s_1, s_2, ..., s_n\}$ be a collection of n items. Each item s_j has a positive integer size k_j and a positive value c_j. Given a knapsack of positive integer size K, find a subset of items $S' \subseteq S$ such the total size of the items in S' is $\leq K$ and the total value is maximized.*

We will use 2-dimensional induction on (n, K) to solve the problem.

B.C.: If either $n = 0$ or $K = 0$, the solution to the problem is obviously the empty set ϕ with the maximized value 0.

I.H.: Now assume that we can solve all versions of the knapsack problem with with an input less than (n, K). In other words, we are talking about the knapsack problems $\{0, 1, 2, ..., n\} \times \{0, 1, 2, ..., K\} - (n, K)$. (See the table below.) Note that this is $(n + 1)(K + 1) - 1$ problem scenarios, and obviously to store the solutions to all of these problems scenarios, we will need an $(n + 1) \times (K + 1)$ table.

	Knapscak size : K					
Number of Items : n	0	1	2	$\cdots$	$K-1$	K
ϕ	✓	✓	✓	$\cdots$	✓	✓
$\{s_1\}$	✓	✓	✓	$\cdots$	✓	✓
$\{s_1, s_2\}$	✓	✓	✓	$\cdots$	✓	✓
$\vdots$	✓	✓	✓	$\cdots$	✓	✓
$\{s_1, s_2, ..., s_{n-1}\}$	✓	✓	✓	$\cdots$	✓	✓
$\{s_1, s_2, ..., s_{n-1}, s_n\}$	✓	✓	✓	$\cdots$	✓	?

I.S.: The question here is given the nK solutions from the I.H., can we use these solutions to solve when we are given n items and a knapsack of size K?

Consider the item s_n. Observe that there are only two possibilities, either s_n is in the optimal solution or not.

If we assume s_n is in the optimal solution, then it will take up k_n space in the knapsack, and we have to fill in the remain portion of the knapscak $K - k_n$ optimally. Thus, if s_n is in the knapsack, the maximum value will be $c_n + C(n - 1, K - k_n)$.

On the other hand, if s_n is not in the knapsack, then we will to fill up the entire knapsack using the other $n - 1$ items. The maximum value will be $C(n - 1, K)$.

Obviously, the optimal solution will have a value $\max\{c_n + C(n - 1, K - k_n), C(n - 1, K)\}$.

This immediately yields the following knapsack algorithm:

algorithm: Knapsack

input: An array of n items $s_1, s_2, ..., s_n$. Each item s_j has a positive integer size k_n and a positive value c_j. A knapsack of positive integer size K.

output: A subset of items whose total size if $\leq K$, and whose total value is maximized.

1. Create a table KP of size $(n+1) \times (K+1)$.
2. Set $KP[i,j] = 0$ for all $i, j = 0$. (Note that this is for illustrative purpose. In the acutaly implementation, we can assume the exitence of these entries implicitly.)
3. For $i = 1$ to n do:
4. For $j = 1$ to K do:
5. $KP[i,j] = \max\{c_i + C(i-1, K-k_i), C(i-1, K)\}$.

The running time is obvioulsy $O(nK)$. However, notice that this running time is not a polynomial only in terms of the input size, which is $O(n)$. Therefore, the running time may not be efficeint if $K = \Omega(2^n)$.

10 Line Segment Intersection and Binary Search Tree

Problem 7 *Given a set of n line segments on the plane. Find all of their intersections. (See Figure 6 (a) for illustrations.)*

The algorithm that we will present here is called the "plane sweeping" algorithm. It calculates the intersections from left to right, mimicing how a human will mark the intersections systematically from left to right. (See Figure 6 (a) for illustrations.) In this course, we will look at a simpler version of the problem.

10.1 Intersections of Horizontal and Vertical Line Segments

Problem 8 *Given a set of n horizontal and m vertical line segments on the plane. Find all of intersections between a horitonal and a vertical line segment. (See Figure 6 (b) for illustrations.)*

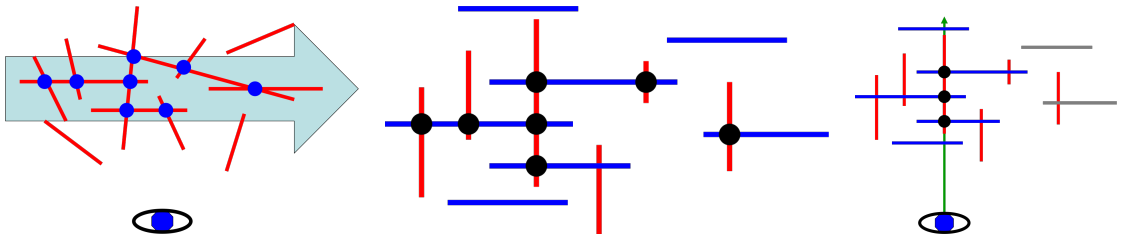

Figure 6:

For ease of explanation, we shall assume:

(1) Each hotizontal line segment is represented by a triple (x_L, x_R, y), where x_L is the $x-$coordinate of the left endpoint, x_R is the $x-$coordinate of the right endpoint, and y is the $y-$coordinate of the line segment.

(2) Each vertical line segment is represented by a triple (x, y_T, y_B), where x is the $x-$coordinate of the line segment, y_T is the $y-$coordinate of the top endpoint, and y_B is the $y-$coordinate of the bottom endpoint.

The key idea of the algorithm is that we will imagine there is a vertical sweepline moving across the plane. The induction hypothesis is that all intersections to the left of the sweepline have been calculated. Clearly, if the sweepline is to the left of the left-most left endpoints of a horizontal line segments, then the I.H., holds true. This is essentially the basis of our algorithm. So, what happens to those vertical line segments to the left of the left-most left endpoints? Obviously, they will not lead to any intersections and should be removed from consideration.

Before we consdier the induction step, i.e., how to move the sweepline to its right, let's first consider:

(1) Under what situation, will the sweepline bump into an intersection?

(2) Once we find out that the sweepline may bump into an intersection, how do we calculate the intersection(s)?

To answer the firt question, observe that the sweepline may "bump" into an intersection only when it coincides with a vertical line segment. This immediately implies that we can sort the vertical line segments based on their $x-$coordinates from left to right, and pause the sweepline every time it coincides with a vertical line segment.

To answer the second question, once our sweepline coincides with a vertical line segment, to find all the intersections on it, we need to know which horizontal line segments are intersecting the sweepline. (See Figure 6 (c) for illustrations.) As the sweepline moves from left to right, this "collection" of horizontal line segments has to be maintained, since new line segment may be inserted, while some line segment may be deleted. The data structures that we have learned so far that can handle efficeint insertion and deletion is the BST, therefore, we will use a BST.

From the above discussion, we have the following algroithmic framework. This is an **event-driven** algorithm that only pause the moving sweepline when certain **event of interest** happen.

Algorithm: Plane sweeping

Input: n horizontal and m vertical line segments

Output: All intersections from a horizontal and a vertical line segment.

1. Sort the $x-$coordinates of the left endpoints, right endpoints, and the vertical line segments and placed all these **events** in an array E of size $2n + m$.
2. Create a BST T.
3. For $j = 1$ to $2n + m$ do:
4. Case 1: If $E[j]$ is the left endpoint of a horizontal line segment, insert the horizontal line segment into T.
5. Case 2: If $E[j]$ is the right endpoint of a horizontal line segment, delete the horizontal line segment from T.
6. Case 3: if $E[j]$ is the $x-$coordinate of a vertical line segment, calculate the intersections between the vertical line segment.

The key here is how to deal with case 3 in the above algorithm. This requires sovling the **range query** problem below:

Problem 9 *Given a BST T, and an interval $[l, r]$, output all the nodes in T that are within the interval. (See Figure 7 for illustrations.)*

Algorthm: Range Query

Input: A BST T, and a query interval $[l, r]$.

Output: A the keys in T that are within the query interval.

1. Let v_l be the smallest element $\geq l$, and v_r be the largest element $\leq r$.

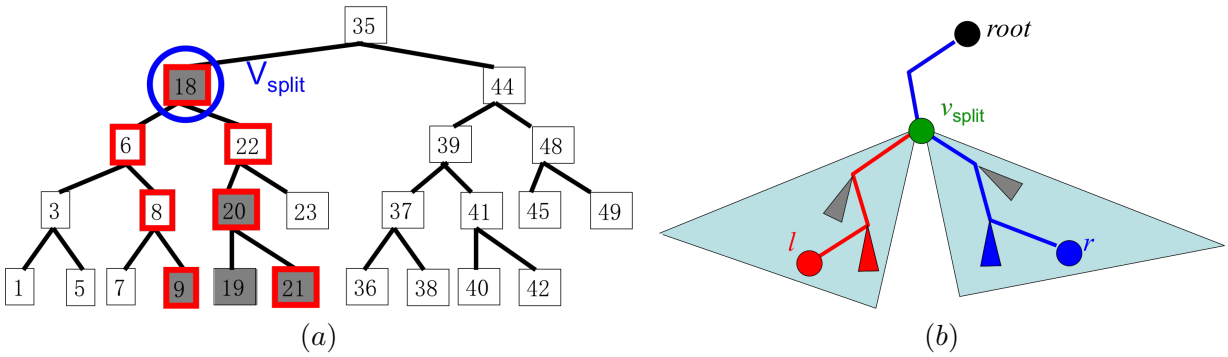

Figure 7:

2. Let v_{split} be the lowest common ancestor of v_l and v_r.
3. For every node v from v_{split} down toward v_l do:
4. if $v.key \geq l$, output $v.key$.
5. Output every keys in the right sub-tree of v.
6. For every node v from v_{split} down toward v_r do:
7. if $v.key \leq r$, output $v.key$.
8. Output every keys in the left sub-tree of v.

The running time of the above algorithm is $O(k + \log n)$.
Here is the complete algorithm:

Algorithm: Plane sweeping

Input: n horizontal and m vertical line segments

Output: All intersections from a horizontal and a vertical line segment.

1. Sort the $x-$coordinates of the left endpoints, right endpoints, and the vertical line segments and placed all these **events** in an array E of size $2n + m$.
2. Create a BST T.
3. For $j = 1$ to $2n + m$ do:
4. Case 1: If $E[j]$ is the left endpoint of a horizontal line segment, insert the horizontal line segment into T.
5. Case 2: If $E[j]$ is the right endpoint of a horizontal line segment, delete the horizontal line segment from T.

6. Case 3: if $E[j]$ is the $x-$coordinate of a vertical line segment, use range query with its $[y_B, y_T]$ to calculate and output the intersections.

The running time of the algorithm is $O((m+n)\log n + k)$.