# Algorithms

Selected Lecture Notes

Partha Bhowmick
IIT Kharagpur

# Contents

# Chapter 1

# Search Trees

The figure aside shows a sculpture by Andrew Rogers in Jerusalem. The sculpture ratio is proportioned according to ***Fibonacci numbers***. Interestingly, it has a subtle connection with the ***golden ratio***, which is again connected with search trees (Sec. 1.2).

## 1.1 Binary Search Tree (BST)

Definition 1.1 (Binary Tree) *A binary tree is a data structure in the form of a* **rooted tree** *in which each node has at most two children. A recursive definition: A binary tree is either empty or it contains a root node together with two binary trees called the left subtree and the right subtree of the root.*

Definition 1.2 (BST) *A BST is either empty or a binary tree with the following properties:*

*(i) All keys (if any) in the left subtree of the root precede the key in the root.*

*(ii) The key in the root precedes all keys (if any) in the right subtree of the root.*

*(iii) The left and the right subtrees of the root are BST.*

C declaration of a binary tree or a BST is as follows.

```
typedef struct tnode {
    int x; //info
    struct tnode *left, *right;
} node;
```

### 1.1.1 Traversal in a Binary Tree

Let $u$ be any node and $L$ and $R$ be its respective left and right subtrees. Then the following three types of traversals are defined on a binary tree or a BST:

***Inorder traversal:*** Represented as $LuR$, the left subtree $L$ is (recursively) visited (and nodes reported/processed) first, then the node $u$, and finally the right subtree $R$.

***Preorder traversal:*** $uLR$.

***Postorder traversal:*** $LRu$.

Note that, by inorder traversal of a BST, we always get the sorted sequence of its keys. C-snippet for inorder traversal is given below. For other two types, it's obvious.

```
void in_order(node *u)
{
    if (u!=NULL){
        in_order(u->left);
        visit(u); //ex: print(u)
        in_order(u->right);
    }
}
```

### 1.1.2 Searching in a BST

C-snippet for searching is as follows.

```
main()
{
    node *root;
    int k;
    ...
    p = bst_search(root, k);
    ...
}

node *bst_search(node *p, int k)
{
    if (p!=NULL)
        if (k < p->x)
            p = bst_search(p->left, k);
        else if (k > p->x)
            p = bst_search(p->right, k);

    return p;
}
```

The pseudo-code to search a key $k$ in a BST is as follows:

```
Algorithm SEARCH-BST($r, k$)
1. if $k < key[r]$
2.     SEARCH-BST($left[r], k$)
```

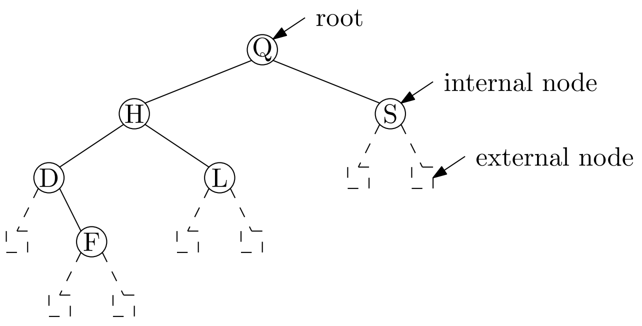


Figure 1.1: An extended BST with $n = 6$ internal nodes and $n + 1 = 7$ external/dummy nodes. It has $I(n) = 1 \times 2 + 2 \times 2 + 3 \times 1 = 9$ and $E(n) = 2 \times 2 + 3 \times 3 + 4 \times 2 = 21$. Hence, for this particular BST, we get $S(n) = (9 + 6)/6 = 2.50$ and $U(n) = 21/7 = 3.00$.

```
3. else if $k > key[r]$
4.     SEARCH-BST($right[r], k$)
5. return $r$
```

## Time complexity

### Best case:

$T(n) = O(n)$, since the search key $k$ may be the very root key.

### Worst case:

$T(n) = T(n - 1) + O(1) = O(n)$, which arises when the BST is fully skewed and the search terminates at the bottommost leaf node.

### Average case:

We define the following terms for a BST having $n$ nodes (see Fig. 1.1):

*Internal path length*: $I(n)$ = Sum of path lengths of all internal nodes from the root (of BST).
*External path length*: $E(n)$ = Sum of path lengths of all external (dummy) nodes from the root.
*Average number of comparisons for successful search*: $S(n)$.
*Average number of comparisons for unsuccessful search*: $U(n)$.

Observe that

$$S(n) = \frac{I(n) + n}{n}, \quad U(n) = \frac{E(n)}{n + 1}. \tag{1.1}$$

It can be proved by induction that

$$E(n) = I(n) + 2n. \tag{1.2}$$

Hence,

$$U(n) = \frac{I(n) + 2n}{n + 1}. \tag{1.3}$$

Eliminating $I(n)$ from Eqns. 1.1 and 1.3,

$$nS(n) = (n + 1)U(n) - n,$$

$$\text{or,} \quad S(n) = \left(1 + \frac{1}{n}\right) U(n) - 1. \tag{1.4}$$

To find the *average* number of comparisons for successfully searching *each* of the $n$ keys, we consider its *insertion order*. If a key $x$ was inserted as the $i$th node, namely $u_i$, then the average number of comparisons for its unsuccessful search in that instance of the tree containing the preceding $(i-1)$ nodes, is given by $U_{i-1}$. Once it's inserted, we can successfully search for it and the search terminates at the node $u_i$, which was a dummy node where its unsuccessful search terminated. Hence, $S_i = U_{i-1} + 1$, as one extra comparison is required for the successful search terminating at $u_i$ compared to the unsuccessful search terminating at the dummy node located at the same position. We estimate the average number of comparisons for all these $n$ nodes based on their insertion orders. Thus, we get

$$S(n) = \frac{1}{n}\sum_{i=1}^{n}(U_{i-1} + 1), \tag{1.5}$$

From Eqns. 1.4 and 1.5,

$$(n+1)U(n) = 2n + U(0) + U(1) + \ldots + U(n-1).$$

To solve the above recurrence, we substitute $n-1$ for $n$ to get

$$nU(n-1) = 2(n-1) + U(0) + U(1) + \ldots + U(n-2).$$

Subtracting the latter from the former,

$$U(n) = U(n-1) + \frac{2}{n+1}. \tag{1.6}$$

Since $U(0) = 0$, we get $U(1) = \frac{2}{2}$, $U(2) = \frac{2}{2} + \frac{2}{3}$, $U(3) = \frac{2}{2} + \frac{2}{3} + \frac{2}{4}$, and so on, resulting to

$$U(n) = 2\left(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n+1}\right) = 2H_{n+1} - 2 \approx 2\ln n = (2\ln 2)\log_2 n, \tag{1.7}$$

where, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$ is the $n$th *harmonic number*, and approximately equals $\ln n$.

Hence, from Eqn. 1.4, we can find $S(n)$, and conclude that $S(n) \approx U(n) \approx (2\ln 2)\log_2 n$.

### 1.1.3 Insertion in a BST

For insertion of a new element, we search for that element in the BST. It's an unsuccessful search, terminating at a dummy node. A new node is created at the position of that dummy node with necessary pointer adjustments. The C-snippet is given below.

```
main()
{
    node *root, *u;
    int k;
    ...
    if ((u = (node *) malloc(sizeof(node))) == NULL)
        error("Exhausted memory.");
    else {
        u->x = k;
        u->left = u->right = NULL;
    }
```

```
        u = bst_insert(root, u);
        ...
    }

    node *insert(node *r, node *u)
    {
        if (r == NULL)
            r = u;
        else if (u->x < r->x)
            r->left = bst_insert(r->left, u);
        else if (u->x < r->x)
            r->right = bst_insert(r->right, u);
        else
            error("key already exists in BST.");
        return r;
    }
```

#### Time complexity

The best-, worst-, and average-case time complexities for insertion of a node in a BST are all similar as those for (unsuccessful) searching.

### 1.1.4 Deletion from a BST

While deleting an element $k$ from a BST, we first search for $k$ in the BST, and then delete the corresponding node, say $u$, based on the following possible cases (see Fig. 1.2).

**Case 1:** If $u$ is a leaf node, then we simply free $u$ and reassign its parent's child pointer (left or right, as applicable) to Null.

**Case 2:** If $u$ has only one subtree, say, the left subtree rooted at $v$, then we reassign the pointer from its parent $p$ to $v$ (and free $u$).

**Case 3:** If $u$ has both the subtrees, the left ($T_2$) rooted at $v$ and the right ($T_3$) at $w$, then we traverse $T_3$ until we reach its leftmost node, say $u'$, which has no left child. After deletion of $u$, the node $v$ should be the inorder predecessor of $u'$. Hence, the left-child pointer from $u'$ is set to $v$ and the pointer from $p$ to $u$ is reset to $w$ (and $u$ is freed).

#### Time complexity

Pointer adjustments in all three cases need constant time. So, the best-, worst-, and average-case time complexities for deletion of a node from a BST are all similar as those for (successful) searching.

## 1.2 AVL Tree

Definition 1.3 (AVL tree) *An AVL tree*[1] *is a* ***self-balancing binary search tree (BST)*** *in which the heights of the two child subtrees of any node differ by at most one.*

[1]Named after its two inventors, G.M. Adelson-Velskii and E.M. Landis (1962): *An algorithm for the organization of information*, Proceedings of the USSR Academy of Sciences **146**: 263–266 (Russian). English translation by Myron J. Ricci in Soviet Math. Doklady, **3**: 1259–1263, 1962.

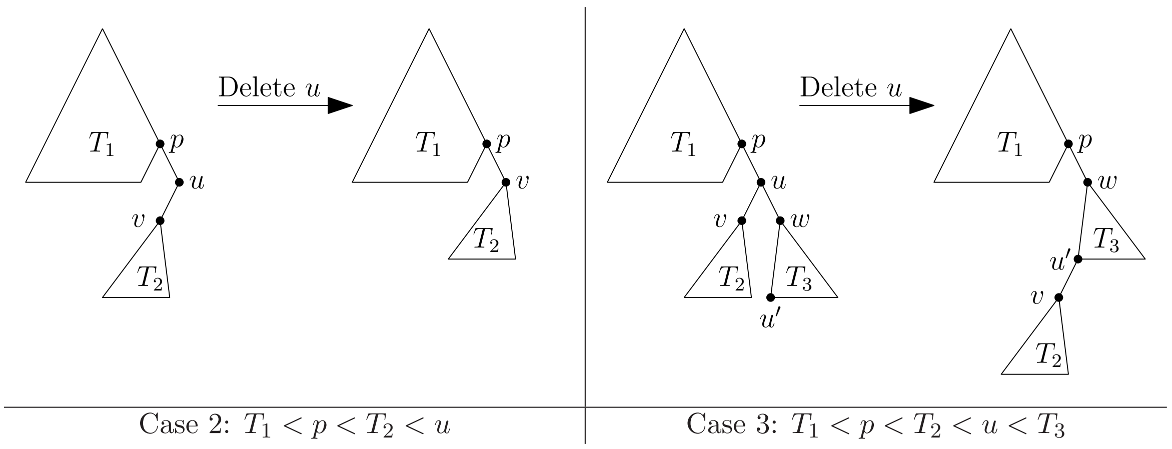

Delete $u$
$T_1$
$p$
$u$
$v$
$T_2$
$w$
$T_3$
$u'$
Case 2: $T_1 < p < T_2 < u$
Case 3: $T_1 < p < T_2 < u < T_3$

Figure 1.2: Deletion from a BST: Case 2 and Case 3.

In an AVL tree having $n$ nodes, searching, insertion, and deletion all take $O(\log n)$ time in both the average and the worst cases. Insertions and deletions may require the tree to be re-balanced by one or more tree rotations. The ***balance factor*** of a node is the height of its left subtree minus the height of its right subtree (a convention, although the opposite is equally valid). A node with the balance factor 0 or $\pm 1$ is said to be ***balanced***. A node with any other balance factor is considered unbalanced and requires re-balancing. The balance factor is either stored directly at each node or computed from the heights of the subtrees (how!?). In short, a BST is an AVL tree iff each of its nodes is balanced.

AVL trees are often compared with red-black trees because they support the same set of operations and because red-black trees also take $O(\log n)$ time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications.[1]

### 1.2.1 Worst-case AVL (Fibonacci) trees

We find the minimum number of nodes, say $n_h$, in an AVL tree of height $h$. Let $h$ be the height of an AVL tree. Then, for the minimality of its node count, we have

$$n_h = n_{h-1} + n_{h-2} + 1, \tag{1.8}$$

where $n_0 = 1$ and $n_1 = 2$, since the minimality of node count demands a similar (recursive) minimality of the subtrees.

By adding 1 to both sides of the above equation, we get the Fibonacci relation

$$F_k = F_{k-1} + F_{k-2}, \quad \text{where } F_0 = 0 \text{ and } F_1 = 1, \tag{1.9}$$

so that $F_k = n_h + 1$, whereby $F_3 (= 2)$ in starting correspondence with $n_0 + 1 (= 2)$.

To solve the above recurrence, we use the following ***generating function*** with the Fibonacci numbers as coefficients:

$$\begin{aligned}
F(x) &= F_0 + F_1 x + F_2 x^2 + \ldots + F_n x^n + \ldots \\
xF(x) &= \phantom{F_0 +} F_0 x + F_1 x^2 + \ldots + F_{n-1} x^n + \ldots \\
x^2 F(x) &= \phantom{F_0 + F_0 x +} F_0 x^2 + \ldots + F_{n-2} x^n + \ldots
\end{aligned}$$

[1] Ben Pfaff (June 2004). *Performance Analysis of BSTs in System Software*, Stanford University. See `http://www.stanford.edu/~blp/papers/libavl.pdf`.

or, $(1 - x - x^2)F(x) = F_0 + (F_1 - F_0)x = x$,
or,

$$F(x) = \frac{x}{1 - x - x^2} = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi x} - \frac{1}{1 - \psi x}\right), \tag{1.10}$$

where $\phi = \frac{1}{2}(1 + \sqrt{5})$ and $\psi = \frac{1}{2}(1 - \sqrt{5})$ are the roots of $1 - x - x^2 = 0$. Hence,

$$F(x) = \frac{1}{\sqrt{5}}\left(1 + \phi x + \phi^2 x^2 + \ldots - 1 - \psi x - \psi^2 x^2 - \ldots\right). \tag{1.11}$$

Observe that from the generating function $F(x)$, the coefficient of $x_n$ is $F_n$, which implies

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \psi^n\right) \approx \frac{1}{\sqrt{5}}\left(1.618^n - (-0.618)^n\right) = int\left(\frac{\phi^n}{\sqrt{5}}\right). \tag{1.12}$$

Note: The term $\phi \approx 1.6180339887498948482...$ is called the ***golden ratio***—a figure where mathematics and arts concur at! Some of the greatest mathematical minds of all ages, from Pythagoras and Euclid in ancient Greece, through the medieval Italian mathematician Leonardo of Pisa and the Renaissance astronomer Johannes Kepler, to present-day scientific figures such as Oxford physicist Roger Penrose, have spent endless hours over this simple ratio and its properties. But the fascination with the Golden Ratio is not confined just to mathematicians. Biologists, artists, musicians, historians, architects, psychologists, and even mystics have pondered and debated the basis of its ubiquity and appeal. In fact, it is probably fair to say that the Golden Ratio has inspired thinkers of all disciplines like no other number in the history of mathematics.

As a continued fraction: $\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}$

As a continued square root, or infinite surd: $\phi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \ldots}}}}$

In simpler terms, two quantities are in the golden ratio if the ratio of the sum of the quantities to the larger quantity is equal to the ratio of the larger quantity to the smaller one; that is, $\phi = \frac{a}{b} = \frac{a+b}{a}$.

Synonyms used for the golden ratio are the golden section (Latin: sectio aurea), golden mean, extreme and mean ratio (by Euclid), medial section, divine proportion (Leonardo da Vinci's illustrations), divine section, golden proportion, golden cut, golden number, and mean of Phidias[1].

One of the many interesting facts about golden ratio: A pyramid in which the apothem (slant height along the bisector of a face) is equal to $\phi$ times the semi-base (half the base width) is called a *golden pyramid.* Some Egyptian pyramids are very close in proportion to such mathematical pyramids.

### 1.2.2 Insertion and Deletion from an AVL Tree

Insertion of a node $u$ in an AVL tree consists of two stages: (i) insertion of $u$ as in the case of a BST; (ii) applying single (L or R) or double (LR or RL) rotation at a node where the balance factor is violated (i.e., has become $+2$ or $-2$ after inserting $u$). Rotations are explained in Fig. 1.3

Interestingly, it can be shown that—by considering case-wise possibilities—at most one (single or double) rotation will ever be done. If so, then it's near the newly inserted node $u$.

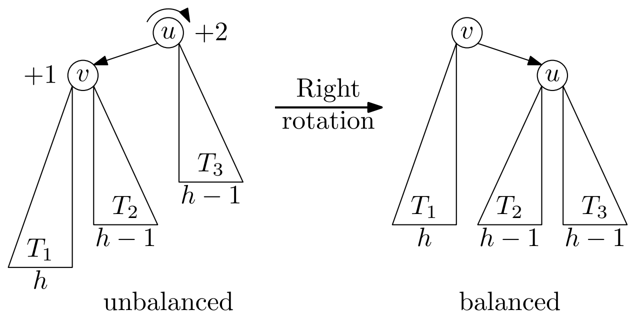
As the height of an AVL tree is $O(\log n)$, insertion takes $O(\log n)$ time for stage (i) and $O(1)$ time for stage (ii), making it $O(\log n)$ in total.

[1]For interesting facts and figures, see `http://en.wikipedia.org/wiki/Golden_ratio`

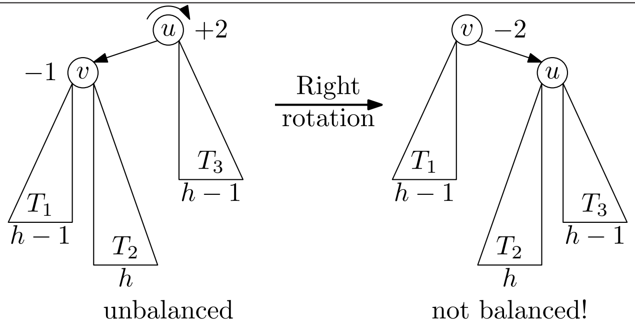
Deletion also consists of two stages as in the case of insertion. The first stage is again similar to that of BST. For the second stage, we have to traverse up to the very root of the BST, go on updating the balance factors of the predecessors along the path to the root, and apply rotations as and when required. The total time complexity for the two stages is, therefore, $O(\log n) + O(\log n) = O(\log n)$.

## Suggested Book

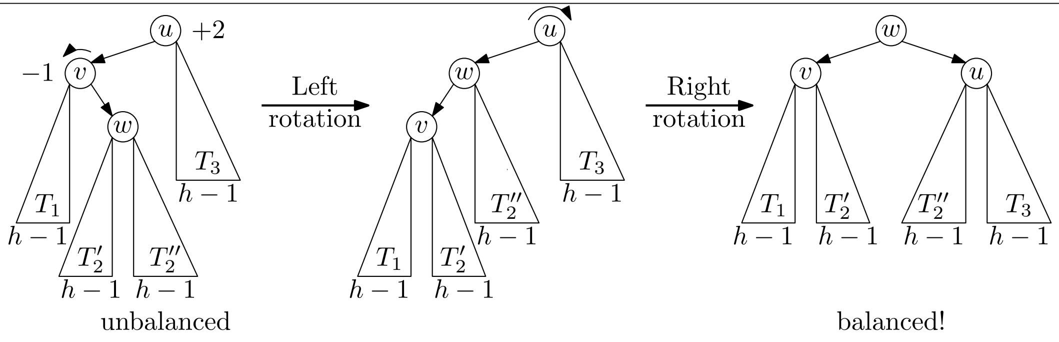
R.L. Kruse, B.P. Leung, C.L. Tondo. *Data Structures and Program Design in C.* PHI, 2000.



***Single right (R) rotation:*** Applied when the node $u$ has balance factor $f = +2$ and its left child $v$ has $f = +1$. Note that the BST property $T_1 \le v \le T_2 \le u \le T_3$ is preserved after rotation.



A case where single right (R) rotation fails: When the node $u$ has balance factor $f = +2$ and its left child $v$ has $f = -1$, although the BST property $T_1 \le v \le T_2 \le u \le T_3$ is preserved after rotation. Double right (LR) rotation balances it correctly.



***Double right (LR) rotation:*** Applied when the node $u$ has balance factor $f = +2$ and its left child $v$ has $f = -1$. Note that the BST property $T_1 \le v \le T_2' \le w \le T_2'' \le u \le T_3$ is preserved after rotation.

Figure 1.3: Single right (R) and double right (LR) rotations for balancing an AVL tree. The other two types of rotations, namely single left (L) and double left (RL) rotations, are similar (to be precise, just vertical reflections of R and LR).

# Chapter 2

# Sorting

"Just as the young squirrel must learn what is climbable and foraging during the dependency period may provide that experience. Beside the selection of food, other feeding behavior such as the (sorting and) opening of nuts, improves with time."—J.P.Hailman

## 2.1 Insertion Sort

► Principle: Iteratively sorts the first $i$ ($2 \leq i \leq n$) elements in a list $L$ using the results of the already-sorted first $i-1$ elements in the previous iteration. To do so, it simply inserts the $i$th element properly among the previously sorted $i-1$ elements.

```
Algorithm INSERTION-SORT($L, n$)
1. for $i \leftarrow 2$ to $n$
2.     $x \leftarrow L[i]$
3.     $j \leftarrow i - 1$
4.     while $j > 0 \wedge L[j] > x$
5.         $L[j+1] \leftarrow L[j]$
6.         $j \leftarrow j - 1$
7.     $L[j+1] \leftarrow x$
```

### 2.1.1 Time complexity

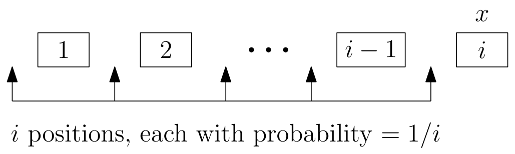
► Best case: Minimum number of comparisons to insert the $i$th ($i > 1$) element $= i - 1$. Hence, $T(n) = (n-1) \cdot O(1) = O(n)$.
► Worst case: Maximum number of comparisons to insert the $i$th ($i > 1$) element $= i - 1$. Hence,

$$T(n) = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2).$$

► Average case: The $i$th element $x$ is equally likely to be placed at any one of the $i$ positions. So, each position has probability $= 1/i$.

Thus, average number of comparisons to insert $x$ is

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



Note: If $x$ is inserted before or after the very first element, then we need $i-1$ comparisons (see figure aside).

Considering all $n$ passes,

$$\begin{aligned} T(n)=\sum_{i=2}^{n}\left(\tfrac{1}{2}(i+1)-\tfrac{1}{i}\right) &= \tfrac{n^2}{4}+\tfrac{3n}{4}-1-\sum_{2}^{n}\tfrac{1}{i} \\ &= O(n^2)+O(n)-O(\log n) \quad \left(\text{since } \sum_{2}^{n}\tfrac{1}{i}=O(\ln n)\right) \\ &= O(n^2). \end{aligned}$$

## 2.2 Quicksort

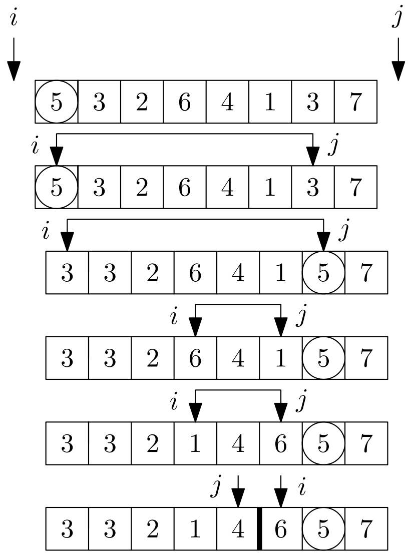
▶ Principle: Based on *divide and conquer*, in-place[1] but not stable[2].

***Divide:*** The input list $L[1..n]$ is partitioned into two nonempty sublists, $L_1 := L[1..q]$ and $L_2 := L[q+1..n]$, such that no element of $L_1$ exceeds any element of $L_2$. The index $q$ is returned from the PARTITION procedure.

***Conquer and combine:*** $L_1$ and $L_2$ are recursively quicksorted in place so that their final combination is sorted.[3][4]

```
Algorithm QUICKSORT($L, p, r$)
1.  if $p < r$
2.      $q \leftarrow$ PARTITION($L, p, r$)
3.      QUICKSORT($L, p, q$)
4.      QUICKSORT($L, q+1, r$)

Procedure PARTITION($L, p, r$)
1.  $x \leftarrow L[p]$
2.  $i \leftarrow p - 1$
3.  $j \leftarrow r + 1$
4.  while TRUE
5.      do $j \leftarrow j - 1$
6.          till $L[j] > x$
7.      do $i \leftarrow i + 1$
8.          till $L[i] < x$
9.      if $i < j$
10.         SWAP($L[i], L[j]$)
11.     else return $j$
```



[1]An in-place sorting needs no extra array.
[2]A stable sorting preserves the relative order of records with equal keys.
[3] "An Interview with C.A.R. Hoare". Communications of the ACM, March 2009 (premium content).
[4]R. Sedgewick, Implementing quicksort programs, Comm. ACM, 21(10):847–857, 1978.

The correctness of the partition algorithm is based on the following facts: (i) $|L_1| > 0$ and $|L_2| > 0$; (ii) $L_1 \cup L_2 = L$; (iii) $L_1 \leq x \leq L_2$;

The correctness of the sorting algorithm follows from inductive reasoning. For one element, the algorithm leaves the data unchanged; otherwise it produces the concatenation of $L_1$ and $L_2$, which are themselves recursively sorted by the inductive hypothesis.

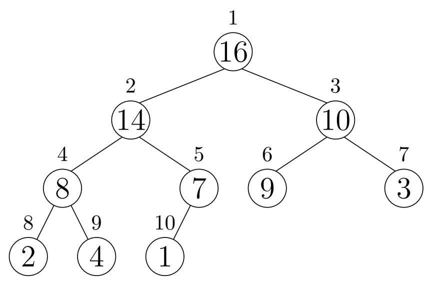
### 2.2.1 Time complexity

▶ Best case: Partition takes $\Theta(n)$ time. So, best-case time complexity of quicksort is $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$.
▶ Worst case: Arises when $\max(|L_1|, |L_2|) = n - 1$ in every step of the recursion, giving $T(n) = T(n-1) + \Theta(n)$, or, $T(n) = \Theta(n^2)$.
▶ Average case: Based on the assumption that the pivot $x$ is equally likely to be the $i$th min for $i = 1, 2, \ldots, n$. If $x$ is the 1st min, then $x$ is the sole element in $L_1$ so that $|L_1| > 0$ (invariant of Partition); for all other cases, $x \in L_2$. So, $|L_1| = 1$ occurs when $x$ is the 1st or the 2nd min. Thus, $Prob(|L_1| = 1) = 2/n$, and $Prob(|L_1| = q) = 1/n$ for $q = 2, \ldots, n-1$. Hence, the worst-case time complexity of quicksort is

$$\begin{aligned}
T(n) &= \frac{1}{n}\left((T(1) + T(n-1)) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) \qquad (2.1)\\
&\leq \frac{1}{n}\left(O(n^2) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n)\\
&= \frac{1}{n}\left(\sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n), \quad \text{as } \tfrac{1}{n}O(n^2) = O(n) \text{ and } O(n) + \Theta(n) = \Theta(n)\\
&\leq \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n) \qquad (2.2)
\end{aligned}$$

We solve the above equation using the method of substitution (induction), with the hypothesis that $T(q) \leq aq \log q$ for a suitable constant $a > 0$ and $\forall\ q < n$. Then,

$$\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{q=1}^{n-1} aq \log q + \Theta(n) \quad = \frac{2a}{n}\sum_{q=1}^{n-1}(q \log q) + \Theta(n)\\
&= \frac{2a}{n}\left(\sum_{q=1}^{\lceil n/2\rceil - 1}(q \log q) + \sum_{q=\lceil n/2\rceil}^{n-1}(q \log q)\right) + \Theta(n)\\
&\leq \frac{2a}{n}\left(\log(n/2)\sum_{q=1}^{\lceil n/2\rceil - 1} q \ + \ \log n \sum_{q=\lceil n/2\rceil}^{n-1} q\right) + \Theta(n)\\
&= \frac{2a}{n}\left((\log n - 1)\sum_{q=1}^{\lceil n/2\rceil - 1} q \ + \ \log n \sum_{q=\lceil n/2\rceil}^{n-1} q\right) + \Theta(n)\\
&= \frac{2a}{n}\left(\log n \sum_{q=1}^{n-1} q \ - \ \sum_{q=1}^{\lceil n/2\rceil - 1} q\right) + \Theta(n)
\end{aligned}$$



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

Figure 2.1: Top: A heap as a complete binary tree. Bottom: The corresponding array $A$.

$$
\begin{aligned}
&= \frac{2a}{n}\left(\frac{1}{2}n(n-1)\log n \;-\; \frac{1}{2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)\left\lceil\frac{n}{2}\right\rceil\right) + \Theta(n)\\
&\leq \frac{2a}{n}\left(\frac{1}{2}n(n-1)\log n \;-\; \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2}\right) + \Theta(n)\\
&= a(n-1)\log n - \frac{a}{2}\left(\frac{n}{2} - 1\right) + \Theta(n)\\
&= an\log n - \left(\frac{a}{2}\left(\frac{n}{2} - 1\right) + a\log n - \Theta(n)\right),
\end{aligned}
$$

which can be made smaller than $an\log n$ for a sufficiently large value of $a$, $\forall\ n \geq n_0$. Thus, $T(n) = O(n\log n)$.

## 2.3 Heapsort

A (binary max) ***heap*** is a ***complete binary tree***, i.e., its levels are completely filled except possibly the lowest, which is filled from left to right (*shape property of heap*). Further, (the key at) each of its nodes is greater than or equal to each of its children (*heap property*). It is represented as 1-D array, $A$, for a simple-yet-efficient implementation as follows (see Fig. 2.1):

Root of the heap $= A[1]$; for each node corresponding to $A[i]$, the parent is $A[\lfloor i/2 \rfloor]$, left child is $A[2i]$, and right child is $A[2i+1]$. Thus, for a heap having $n$ nodes, $A[\lfloor i/2 \rfloor] \geq A[i]$ for $2 \leq i \leq n$, by heap property; and $A[1]$ contains the largest element. Heapsort uses the heap and its properties most efficiently for sorting[1].

▶ Principle: Based on *selection principle*, and so in-place but not stable (as quicksort). It first constructs the heap from $A[1..n]$ using the procedure BUILDHEAP (Step 1). It uses two variables: $length[A] = n$ (remains unchanged) and $heapsize[A]$, the latter being decremented as the heap is iteratively reduced in size (Step 2) while swapping the root of (reducing) heap with its last node (Step 3), and then cutting off the last node (current max; Step 4) so as to grow the sorted sublist at the rear end of $A$. The procedure HEAPIFY is used to rebuild the heap (Step 5), which has lost its *heap property* due to the aforesaid swapping.

[1] J. W. J. Williams. Algorithm 232 – Heapsort, 1964, Communications of the ACM 7(6): 347–348.

```
Algorithm HeapSort($A$)
1. BuildHeap($A$)
2. for $i \leftarrow length[A]$ downto 2
3.     Swap($A[1], A[i]$)
4.     $heapsize[A] \leftarrow heapsize[A] - 1$
5.     Heapify($A, 1$)

Procedure BuildHeap($A$)
1. $heapsize[A] \leftarrow length[A]$
2. for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
3.     Heapify($A, i$)
```

```
Procedure Heapify($A, i$)
1. $l \leftarrow$ Left[$i$],
   $r \leftarrow$ Right[$i$]
2. if $l \leq heapsize[A] \wedge A[l] > A[i]$
3.     $largest \leftarrow l$
4. else $largest \leftarrow i$
5. if $r \leq heapsize[A] \wedge A[r] > A[largest]$
6.     $largest \leftarrow r$
7. if $largest \neq i$
8.     Swap($A[i], A[largest]$)
9.     Heapify($A, largest$)
```

### 2.3.1 Time complexity

The height $h$ of a node in the heap is measured from the bottommost level (height 0) of the heap. Hence, the time required to Heapify a node at height $h$ is $O(h)$, since it involves exchanges between at most $h$ nodes and their left or right children. Now, in a heap having $n$ nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$. Hence, the time complexity of BuildHeap is

$$\sum_{h=0}^{\lceil \log n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left( n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right). \tag{2.3}$$

Now, for $|x| < 1$, we have

$$1 + x + x^2 + \ldots + x^h + \ldots = \frac{1}{1-x}.$$

Differentiating and then multiplying both sides by $x$,

$$x + 2x^2 + 3x^3 + \ldots + hx^h + \ldots = \frac{x}{(1-x)^2},$$

$$\text{or,} \quad \sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2},$$

$$\text{or,} \quad \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \quad [\text{putting } x = 1/2],$$

$$\text{or,} \quad O\left( n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right) = O(n). \tag{2.4}$$

Hence, HeapSort takes $O(n \log n)$ times, since BuildHeap takes $O(n)$ time and each of the $(n-1) = O(n)$ Heapify($A, 1$) calls (Step 5 of HeapSort) takes $O(\log n)$ time.

## 2.4 Linear-time Sorting

All the algorithms discussed in the preceding sections are known as ***comparison sorts***, since they are based on comparison among the input elements. Any comparison sort needs $\Omega(n \log n)$ time in the worst case, as explained next.

A comparison sort can be represented by a (binary) ***decision tree*** in which each non-leaf node corresponds to a comparison between two elements, and each leaf corresponds to a permutation of $n$ input elements. Thus, there are $n!$ leaf nodes, out of which exactly one contains the sorted list. Execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to this node. If the height of the decision tree is $h$, then it can contain at most $2^h$ leaves. So,

$$\begin{aligned} & n! \leq 2^h \\ \text{or,} \quad & h \geq \log(n!) > \log\left(\tfrac{n}{e}\right)^n \quad \text{by Stirling's approximation: } n! = \sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n\left(1+\Theta\left(\tfrac{1}{n}\right)\right) \\ \text{or,} \quad & h \geq n\log n - n\log e = \Omega(n\log n). \end{aligned}$$

### 2.4.1 Counting Sort

Applicable when the input elements belong to a set of size $k$. It creates an integer array $A$ of size $k$ to count the occurrence of $A[i]$ in the input, and then loops through $A$ to arrange the input elements in order. For $k = O(n)$, the algorithm runs in $O(n)$ time. Counting sort cannot often be used because $A$ needs to be reasonably small for it to be efficient; but the algorithm is extremely fast and demonstrates great asymptotic behavior as $n$ increases. It can also be used to provide stable behavior.

### 2.4.2 Bucket sort

A divide-and-conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm (e.g., insertion sort), or by recursively applying the bucket sort. It is most effective and runs in linear time on data with limited values (e.g., to sort a million integers ranging from 1 to 1000).

### 2.4.3 Radix sort

Sorts numbers by processing individual digits. It either processes digits of each number starting from the least significant digit (LSD) or from the most significant digit (MSD). The former, for example, first sorts the numbers by their LSD while preserving their relative order using a stable sort; then it sorts them by the next digit, and so on up to their MSD, ending up with the sorted list. Thus, $n$ numbers consisting of $k$ digits can be sorted in $O(nk)$ time, which reduces to $O(n)$ time for a fixed $k$. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable.