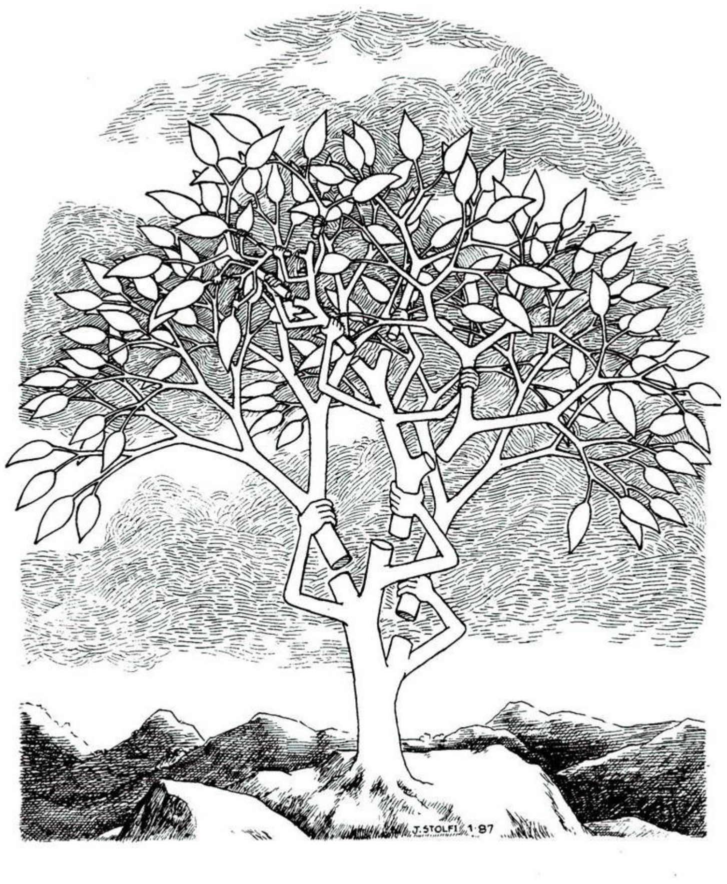
Advanced Data Structures

course notes for CS4700



Yadu Vasudev
yadu@cse.iitm.ac.in

Image credit: A self-adjusting search tree by Jorge Stolfi

Contents

1 Amortized Analysis

While programming in C++ using the Standard Template Library, one of the most common templates that we use is the `vector` class. We use `vector<int>` to refer to a vector of integers and some of the operations that it supports are as follows.

- `v.at(i)`: Returns the element at position `i` in the vector `v`.
- `v.push_back(k)`: Adds the element `k` at the end of the vector `v`.
- `v.pop_back`: Deletes the element at the end of the vector `v`.

You can refer `https://cplusplus.com/reference/vector/vector/` for the member functions of the class.

There are many more operations, but these would suffice for our discussions. Ideally, we would want these operations to be performed in $O(1)$ time. The operations `push_back` and `pop_back` should also change the length of the vector accordingly. While using the `vector` class, you would not have to perform the annoying job of specifying the length of the array beforehand. We can work as though we have an array of as large a size as we want. We know that practically this is impossible. The easiest way that we can think of for obtaining this feature would be to resize an array by copying the content each time it is resized. But, this would mean that you are unlikely to perform `push_back` and `pop_back` in $O(1)$ time. If you try to manage the $O(1)$ bound by using a linked list, say, then the `at(i)` operation will no longer be $O(1)$. So what happens under the hood so that we get good bounds on all the operations?

1.1 Dynamic arrays

We will use standard arrays for the `vector` class, but we will resize the array as the vector grows and shrinks. This will let us obtain an $O(1)$ running time for `at(i)`, but we will lose out on `push_back` and `pop_back`. Instead of getting $O(1)$-worst case running time, we will get something weaker. In fact, a single `push_back` or `pop_back` will incur a worst-case $O(n)$ running time (where n is the size of the array). Nonetheless, we will show that the sum of the running times of m `push_back` operations will only take $O(m)$. Thus, the *amortized cost* of one operation is $O(1)$.

Amortized cost refers to the average cost of a single operation, where the average is over a *worst-case* sequence of some m operations. Note that this is different from the average-case analysis that you might have seen for, say quicksort, where the input is considered to be randomly ordered.

Amortization is typically used when calculating the EMIs on loans. If a person takes x amounts of loan at an interest rate of r for a period of n years, the final amount to be paid is typically divided equall for the entire duration of the loan. The component of interest and principal in EMI payment varies, but the total amount to be paid every month remains unchanged. This is the essence of amortized analysis in algorithms as well. We devise the analysis so that the cost at every step remains the same, but some of the cost is the actual cost at the step and the others are some extra costs that will take care of costlier operations later. In the case of dynamic arrays, the costlier operations are resizings, but we will design the algorithm in such a way that these costlier operations do not occur often, and its cost can be accounted for in the easier operations done earlier.

A naive way to perform resizing is to keep the array always full, and to resize it if a new element comes in. But this is wasteful because you will end up resizing for every insertion. It would be more efficient if we create a larger array when resizing so that a number of consequent insertions can be performed efficiently. Let's start with the case of insertions alone for now.

We start with an array A of some fixed size, say 64. The insertions are performed in $O(1)$ time until this array is full. Once the 65^{th} element arrives, we will resize the array A to double its current size. Even though this resizing is a costly operation, the next 63 inserts will all take only $O(1)$ time, before we need another resize operations. We will assume that the time taken to allocate memory for a new array is $O(1)$. The cost of the resizing will include the time taken to copy each element into the new array. Copying one element takes $O(1)$ time, and hence copying an array of size n takes $O(n)$ time. Thus in the worst-case, the complexity of inserting an element in the array is $O(n)$. We will now show that inserting n elements in such a dynamic array will take $O(n)$-time in total.

The dynamic array A has A.size denoting the number of elements in it currently and A.capacity denoting the maximum number of elements that can be stored in it. During every insertion, we check if $A.\text{size} = A.\text{capcacity}$. If so, we create a new array A' with $A'.\text{capacity} = 2 \cdot A.\text{capacity}$, and copy the contents of A to A'. We will then insert the new element into A'. We will look at three ways to analyze this algorithm. These methods are generic and we will see many more uses of them later.

1.1.1 Aggregate analysis

In aggregate analysis, we sum up the costs of each operation individually. Thus if $t(n)$ is the total time for n operations, we will say that the amortized cost of each operations is $O(t(n)/n)$. Let us try to sum up the costs of n insertions starting from an empty array. Assume that initially, the capacity is 0, and when the first element is inserted, the size is made 2. Now, each time the capacity is some 2^i and a new element is inserted, the size is doubled to 2^{i+1}. Thus the cost of the i^{th} operation t_i is i if $i = 2^k + 1$ for some i, or it is 1 if the size of the array is less than its capacity.

Thus the total cost of the inserting n elements into a dynamic array is given by the expression

$$\sum_{i=1}^{n} t_i = n + \sum_{j=1}^{\log n} 2^j = O(n).$$

Thus the amortized cost of a single insertion is $O(1)$.

1.1.2 Accounting method

In the accounting method, we try to charge each opertion a cost that may be potentially higher than the actual cost of the operation. The additional cost is thought of as some credit that can be used for a later costly operation - in the case of a dynamic array this would be the resizing operation.

Suppose that at a particular stage in the algorithm the size n is exactly half of the capacity of the array - this is the case when a new array has been created. Now, for each insertion, we charge 3 units - one for its insertion, one unit to be used for a future copy of this element, and one unit for copying one of the n elements that were copied. Thus the cost of an insertion is 3 units, which is $O(1)$. We will now show that after inserting n elements, we have enough credit stored such that the copying can be taken care of with these credits.

Notice that when n more elments are inserted, then the total credits saved is $2n$. Now when a new element is about to be inserted, we will use up these $2n$ credits for the copying of the $2n$ elements in the array. Thus the cost of the insertion is just 3 units for the new element that will be inserted in the resized array. Hence the amortized cost of any insertion is $O(1)$.

1.1.3 Potential method

In this method we assign a potential function with the data structure at all points of time, and every operation causes a change in the potential. In the case of the array, we will define a function $\Phi(i)$ to denote the potential associated with the dynamic array after the i^{th} insertion. We will make sure that $\Phi(0) = 0$ and $\Phi(i) \geq 0$ for all i. The intuition is that the potential measures the credit that has been stored during simpler operations, and which can then be used up in a costlier operation. In this sense, it is similar to the accounting method. Most of the examples that we will see using the potential method can also be analyzed using the accounting method, but this may not always be true.

The amortized cost of the i^{th} operation, denoted by $\widehat{t_i}$, is defined as

$$\widehat{t_i} = t_i + \Phi(i) - \Phi(i-1),$$

where t_i is the actual cost of the i^{th} insertion (including the resizing). Now, the goal is to define the potential function $\Phi(i)$ in such a way that during an insertion that includes a resizing, the potential $\Phi(i)$ decreases to a sufficiently small value compared to $\Phi(i-1)$, and consequently $\widehat{t_i}$ is small. In other words, there is a large amount of stored credit/potential (given by $\Phi(i-1)$) that can take care of the high t_i value.

The total cost over n operations will then be

$$\sum_{i=1}^{n} \widehat{t_i} = \sum_{i=1}^{n} t_i + \Phi(n) - \Phi(0).$$

Since $\Phi(n) \geq 0$, we have $\sum_{i=1}^{n} t_i \leq \sum_{i=1}^{n} \widehat{t_i}$ and thus the total amortized cost is an upper-bound on the actual cost of the n operations.

Let's try to define a potential function for the dynamic array insertion. We will be guided by our analysis using the accounting method. For every insertion we added one extra credit to the element that was inserted and one for an element in the first half of the array to pay for their cost of copying. Thus, when the array A is full, the total credits that are stored is A.capacity. With this in mind, we will define $\Phi(i)$ as follows: $\Phi(i) = 2 \cdot A.\mathsf{size} - A.\mathsf{capacity}$.

Observe that when the array A is full, the potential is equal to A.capacity - this means that there is sufficient credit to perform the copying that will have to be done if a new element is inserted. Similarly, right after a resizing the array A is half-full and the potential is 0 - the extra potential has all been used up for copying the contents of the old array into the new array. Since at every step, $A.\mathsf{size} \geq \frac{1}{2} A.\mathsf{capacity}$, we have $\Phi(i) \geq 0$.

While computing the amortized cost $\widehat{t_i}$, we have two cases.

- Insertion of the i^{th} element does not cause a resizing: In this case we have $\Phi(i) = \Phi(i-1) + 2$ since only the size increases by 1 and the capacity remains unchanged. Thus $\widehat{t_i} = t_i + \Phi(i) - \Phi(i-1) = 1 + 2 = 3$.

- Insertion of the i^{th} element causes a resizing: Let A' be the old array that was resized to create the new array A. We have $A.\mathsf{capacity} = 2 \cdot A'.\mathsf{capacity}$. Also since the i^{th} insertion triggered a resizing, we must have $A'.\mathsf{size} = A'.\mathsf{capacity}$. Thus we have

$$\begin{aligned}\Phi(i) - \Phi(i-1) &= 2 \cdot A.\mathsf{size} - A.\mathsf{capacity} - 2 \cdot A'.\mathsf{size} + A'.\mathsf{capacity} \\ &= 2(A'.\mathsf{size} + 1) - 2A'.\mathsf{size} - A'.\mathsf{size} = 2 - A'.\mathsf{size}.\end{aligned}$$

  Thus, $\widehat{t_i} = t_i + \Phi(i) - \Phi(i-1) = A'.\mathsf{size} + 1 + 2 - A'.\mathsf{size} = 3$, since the actual cost $t_i = A'.\mathsf{size}$ to copy all the elements in the old array into the new array.

1.1.4 Dynamic arrays with insertions and deletions

In the previous part we only looked at implementing `push_back` efficiently. If we don't resize the array during the `pop_back` operations, then it might end up with a lot of unused space - consider the case of a sequence of insertions followed a sequence of deletions. Analogous to the case of insertions, we could choose to resize the array by making it smaller and copying all the elements into the smaller array. There are two questions one need to answer here.

- What should be the ratio of A.size to A.capacity to do the resizing?

- How small should the new array be?

One way would be to double the size of the array as before during insertions. During deletions, if the size is at most one-fourth of the total capacity, we will resize it so that the capacity is double of the size. This way we will always maintain the following invariant across insertions and deletions.

$$\frac{1}{4}A.\mathsf{capacity} \leq A.\mathsf{size} \leq A.\mathsf{capacity}.$$

The fraction $\ell(A)$ defined as $\frac{A.\mathsf{size}}{A.\mathsf{capacity}}$ is defined as the load on the array A. The invariant says that the load is always at least $1/4$.

We will analyze this using both the accounting method and the potential method. In the accounting method, for every insertion of an element into an array when the *size is at least half of the capacity*, we will add extra 2 units of credit to be used for copying just like in the case of insertion-only arrays. For the case when $\frac{1}{4}A.\mathsf{capacity} \leq A.\mathsf{size} \leq \frac{1}{2}A.\mathsf{capacity}$, we will only charge for the insertion operation. Similarly, when we delete an element and $\frac{1}{4}A.\mathsf{capacity} \leq A.\mathsf{size} \leq \frac{1}{2}A.\mathsf{capacity}$, we will charge an extra 1 unit of credit to be used up while resizing the array when $\frac{1}{4}A.\mathsf{capacity} \geq A.\mathsf{size}$. Thus the amortized cost is $O(1)$ and the cost of resizing is taken care of by the extra credits given.

If we were to define a suitable potential function Φ, we will try to make sure that right after a resizing the potential should be zero. Similarly, when $A.\mathsf{size} = A.\mathsf{capacity}$, the $\Phi(i)$ should be $A.\mathsf{size}$. Similarly, whenever $A.\mathsf{size} = \frac{1}{4}A.\mathsf{capacity}$, the potenital $\Phi(i)$ should again be $A.\mathsf{size}$ since we need it to copy all the elements to the resized array. With this in mind, we can define our potential corresponding to the array A at some point i in time as follows:

$$\Phi(i) = \begin{cases} 2 \cdot A.\mathsf{size} - A.\mathsf{capacity} & \ell(A) \geq 1/2 \\ \frac{1}{2} \cdot A.\mathsf{capacity} - A.\mathsf{size} & \ell(A) < 1/2. \end{cases}$$

Clearly $\Phi(i) \geq 0$ for all i. Observe that both the conditions stated earlier are satisfied by this definition of the potential function. Analogous to the case of insert, we have multiple cases to verify now for both insertion and deletion. Let A_i denote the array afte the i^{th} operation.

- Insert operation when $\ell(A_i) = 1$: This means that the array is full and the insert operation causes a resizing. Thus, we have the following:

$$\begin{aligned} t_{i+1} &= A_i.\mathsf{size} + 1 \\ \Phi(i) &= A_i.\mathsf{size} \\ \Phi(i+1) &= 2(A_i.\mathsf{size} + 1) - 2 \cdot A_i.\mathsf{capacity}. \end{aligned}$$

  Here $A_i.\mathsf{size} = A_i.\mathsf{capacity}$. Thus, $\widehat{t}_{i+1} = t_{i+1} + \Phi(i+1) - \Phi(i) = 3$.

- Insert operation when $1/2 \leq \ell(A_i) < 1$: In this case we have

$$\begin{aligned} t_{i+1} &= 1, \\ \Phi(i) &= 2 \cdot A_i.\mathsf{size} - A_i.\mathsf{capacity}, \\ \Phi_{i+1} &= 2 \cdot (A_i.\mathsf{size} + 1) - A_i.\mathsf{capacity} \end{aligned}$$

Thus $\widehat{t}_{i+1} = 3$.

- Insert operation when $\ell(A_i) < 1/2$, but $\ell(A_{i+1}) \geq 1/2$: In this case, we again have

$$\begin{aligned} t_{i+1} &= 1, \\ \Phi(i) &= \frac{1}{2} \cdot A_i.\text{capacity} - A_i.\text{size}, \\ \Phi(i+1) &= 2 \cdot (A_i.\text{size} + 1) - A_i.\text{capacity} \end{aligned}$$

We can calculate the amortized cost $\widehat{t}_{i+1}$ as follows.

$$\begin{aligned} \widehat{t}_{i+1} &= 1 + 2 \cdot (A_i.\text{size} + 1) - A_i.\text{capacity} - \frac{1}{2} \cdot A_i.\text{capacity} + A_i.\text{size} \\ &= 3 + 3 \cdot A_i.\text{size} - \frac{3}{2} \cdot A_i.\text{capacity} \\ &< 3, \text{ since } \ell(A_i) < 1/2. \end{aligned}$$

- Finally, if $\ell(A_{i+1}) < 1/2$, then we have

$$\begin{aligned} t_{i+1} &= 1, \\ \Phi(i) &= \frac{1}{2} \cdot A_i.\text{capacity} - A_i.\text{size}, \\ \Phi(i+1) &= \frac{1}{2} \cdot A_i.\text{capacity} - A_i.\text{size} - 1. \end{aligned}$$

Thus $\widehat{t}_{i+1} = t_{i+1} + \Phi(i+1) - \Phi(i) = 0$.

Exercise 1.1. Analyze the amortized complexity of the deletion operation by considering all the cases.

2 Randomization

Randomization is ubiquitious in computer science, and we will see data structures and algorithms that use randomization in clever ways. We will look at very basic probabilistic analysis that will be useful for us in analyzing a few randomized data structures that we will see later on.

2.1 Basic discrete probability

We will recall basic discrete probability, the notion of random variables and expectation that will be required in the probabilistic analysis later on. The basic object of interest in probability is a probability space $\mathscr{P}$ that consists of a tuple $(\Omega, \Pr)$ where Ω is the *sample space* that is discrete in our case, and a *probability* function $\Pr : \Omega \to [0,1]$ that satisfies the property that $\sum_{\omega \in \Omega} \Pr[\omega] = 1$.

The sample space captures all the outcomes of a random experiment. For instance, if the random experiment is the tossing of an unbiased coin, then $\Omega = \{\texttt{Heads}, \texttt{Tails}\}$, and $\Pr[\texttt{Heads}] = \frac{1}{2}$ and $\Pr[\texttt{Tails}] = \frac{1}{2}$. A roll of a standard die corresponds to the sample space $\Omega = \{1,2,3,4,5,6\}$ and the probability function $\Pr[i] = \frac{1}{6}$ for every $i \in \{1,2,3,4,5,6\}$. When studying random experiments, we will be interested in subsets of Ω, which are referred to as *events*. Thus $E \subseteq \Omega$ is an event, and $\Pr[E] = \sum_{\omega \in E} \Pr[\omega]$.

In the random experiment corresponding to the roll of an unbiased die, an example of an event E would be the outcomes where an even number comes when the die is rolled. Thus $E = \{2,4,6\}$, and $\Pr[E] = \frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$. We can combine events just like sets. We will use both the set operations $\cup, cap$ as well as boolean operations $\wedge, \vee$ while combining events. Thus $\Pr[E_1 \vee E_2]$ is the probability of the events E_1 or E_2 occurring, and $\Pr[E_1 \wedge E_2]$ is the probability of both the events E_1 and E_2 occurring.

When studying multiple events and their probabilities, we will often study the *conditional probabilities* of events. Let E_1 and E_2 be two events. The conditional probability of E_1 given that E_2 has occurred will be denoted by $\Pr[E_1|E_2]$ and is defined as

$$\Pr[E_1|E_2] = \frac{\Pr[E_1 \wedge E_2]}{\Pr[E_2]}.$$

Two events E_1 and E_2 are said to be *independent* if

$$\Pr[E_1|E_2] = \Pr[E_1].$$

In other words the occurrence of E_2 does not affect the probability of the occurrence of E_1. Consider the case of rolling two unbiased dice. The samples space Ω for this random experiment is the set $\{(i,j) \mid 1 \le i,j \le 6\}$.

Suppose we look at the event E_1 that both dice roll even numbers, let E_2 be the event that the number on the first die is at least 4. Thus

$$E_1 = \{(i,j) \mid i,j \in \{2,4,6\}\}, \text{ and}$$

$$E_2 = \{(i,j) \mid 4 \le i \le 6, 1 \le j \le 6\}.$$

Hence, we have

$$\begin{aligned}
\Pr[E_1] &= 9/36 = 1/4,\\
\Pr[E_2] &= 18/36 = 1/2,\\
\Pr[E_2|E_1] &= 6/9 = 2/3,\\
\Pr[E_1|E_2] &= 6/18 = 1/3
\end{aligned}$$

We could also calculate $\Pr[E_1]$ using the fact E_1 can be thought of as follows: Roll two dice D_1 and D_2 independently. For D_1, A_1 is the event of D_1 turning up with an even number, and A_2 is the event of D_2 turning up with an even number. Now, $\Pr[E_1] = \Pr[A_1 \wedge A_2]$. Sinc the events A_1 and A_2 are independent, we have $\Pr[A_1 \wedge A_2] = \Pr[A_1]\Pr[A_2] = \frac{3}{6} \cdot \frac{3}{6} = \frac{1}{4}$.

Two events E_1 and E_2 are said to be *disjoint* if $E_1 \cap E_2 = \emptyset$. I.e. the sets that correspond to the events do not intersect. While analyzing the probability of random experiments, we will often need to look at various events, their unions and intersections. Here are a few useful identities.

- **Union bound**: Let $E_1, E_2, \ldots, E_k$ be any k events. Then,

$$\Pr\left[\bigvee_{i=1}^{k} E_i\right] \le \sum_{i=1}^{k} \Pr[E_i]$$

- **Inclusion-Exclusion**: Let $E_1, E_2, \ldots, E_k$ be any k events. Then,

$$\Pr\left[\bigvee_{i=1}^{k} E_i\right] = \sum_{I \subseteq \{1,2,\ldots,k\}, I \neq \emptyset} (-1)^{|I|+1} \Pr\left[\bigwedge_{i \in I} E_i\right].$$

- **Disjoint union**: If the events E_i are pairwise disjoint, i.e. $E_i \cap E_j = \emptyset$ for all $i \neq j$, then it follows from the inclusion-exclusion principle that

$$\Pr\left[\bigvee_{i=1}^{k} E_i\right] = \sum_{i=1}^{k} \Pr\left[E_i\right].$$

- **Union of independent events**: A collection $E_1, E_2, \ldots, E_k$ of events are said to be *mutually independent* if for every subset $I \subseteq \{1,2,\ldots,k\}$, we have

$$\Pr\left[\bigwedge_{i \in I} E_i\right] = \prod_{i \in I} \Pr[E_i].$$

For a collection of mutually independent events, we have

$$\Pr\left[\bigvee_{i=1}^{k} E_i\right] = 1 - \prod_{i=1}^{k}\left(1 - \Pr[E_i]\right).$$

- **Baye's theorem**: Let E_1 and E_2 be two events. The following equation, that follows directly from the definition of conditional probability, is known as Baye's theorem.

$$\Pr[E_1|E_2]\Pr[E_2] = \Pr[E_2|E_1]\Pr[E_1].$$

2.2 Random variables and expectation

The outcome of a randomized algorithm depends on the random choices made by the algorithm during its execution. Studying this outcome amounts to understanding the properties of a *random variable*. A random variable X is a function $X : \Omega \to \mathbb{R}$, from a sample space Ω to the set of real numbers. We can associate events corresponding to a random experiment very naturally with random variables.

A random variable is neither random nor a variable!

If a random variable X takes a value a, we can associate that with the event $E \subseteq \Omega$ such that $\omega \in E$ iff $X(\omega) = a$. We also denote this as

$$\Pr[X = a] = \sum_{\omega, X(\omega)=a} \Pr[\omega].$$

An important class of random variables that we will encounter while analyzing algorithms is the indicator random variable, denoted by I, that maps Ω to the set $\{0,1\}$. For an event $E \subseteq \Omega$, an indicator random variable $I : \Omega \to \{0,1\}$ assigns $I(\omega) = 1$ iff $\omega \in E$. Thus, for such variables, $\Pr[I = 1] = \Pr[E]$ and $\Pr[I = 0] = 1 - \Pr[E]$.

Like in the case of events, we will say that two random variables X and Y are independent if for every α, β in the range of the random variables, we have

$$\Pr[X = \alpha \wedge Y = \beta] = \Pr[X = \alpha] \cdot \Pr[Y = \beta].$$

The *expectation* of a random variable X, denoted by $\mathbb{E}[X]$, is defined as the weighted sum

We will not worry about continuous random variables for now...

$$\mathbb{E}[X] = \sum_{\alpha} \alpha \Pr[X = \alpha],$$

where the sum is over the range of the random variable X. Here are a few properties of expectation that will useful in analyzing random variables.

- If I is an indicator random variable, then $\mathbb{E}[I] = \Pr[I = 1]$. In other words, the expected value if the random variable is exactly the probability of the event associated with the random variable occurring.

- If X is a random variable that takes positive integer values, then we can write

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i\geq 1} i \cdot \Pr[X = i], \\ &= \sum_{i\geq 0} \Pr[X \geq i].\end{aligned}$$

- **Linearity of expectation**: Let X_1 and X_2 be two random variables. Then,

  $$\mathbb{E}[X_1 + X_2] = \mathbb{E}[X_1] + \mathbb{E}[X_2].$$

  This identity holds for *any two random variables*, not necessarily independent.

Perhaps the most important identity about expectation that you should keep in mind. This straightforward identity is extremely useful.

- **Conditional expectation**: For a random variable X and an event A, the expected value of X conditioned on the occurrence of the event A is given by

  $$\mathbb{E}[X \mid A] = \sum_{\alpha} \alpha \Pr[X = \alpha \mid A].$$

  For two random variables X and Y, we can write

  $$\mathbb{E}[X] = \sum_{\beta} \mathbb{E}[X \mid Y = \beta] \Pr[Y = \beta].$$

Consider the following simple game between two players R and M. They have an unbiased coin. R tosses the coin and if it is heads then R gets ₹1. Let us calculate the expected amount that R wins if they play the game for 100 rounds. Here the sample space Ω is set of all sequence of length 100 consisting of the characters H and T. The random variable X that captures the amount that R wins his a function from this Ω to the set $\{1, 2, \ldots, 100\}$. We are interested in calculating $\mathbb{E}[X]$.

To compute $\mathbb{E}[X]$, we can decompose X as a sum of indicator random variables and use the linearity of expectation. Let I_j denote the indicator random variable that is 1 if the j^{th} toss of the coin returns a head. From before, we know that $\mathbb{E}[I_j] = 1/2$. The random variable $X = \sum_{j=1}^{100} I_j$. By linearity of expectation,

$$\mathbb{E}[X] = \sum_{j=1}^{100} \mathbb{E}[I_j] = 50.$$

2.3 Randomized Quicksort

Let us recall the quicksort algorithm. It is a prime example of a fast sorting algorithm based on the principle of divide-and-conquer. We have an array A of n elements from a universe $\mathcal{U}$ that have a total order on them. The standard quicksort algorithm first chooses a pivot arbitrarily (say the last element in the array), and then partitions the array A into two arrays, one which has elements lesser than the pivot, and another that has elements greater than the pivot. These two arrays are then recursively sorted.

The partitioning using the pivot can be performed in $O(n)$-time using the PARTITION algorithm given as Algorithm 2.1.

Algorithm 2.1: PARTITION$(A, \texttt{low}, \texttt{high})$

```
pos ← low − 1
for $i$ ← low to high − 1 do
    if $A[i] < A[\texttt{high}]$ then
        pos ← pos + 1
        Swap $A[i]$ and $A[\texttt{pos}]$
Swap $A[\texttt{pos}+1]$ and $A[\texttt{high}]$
return pos + 1 //Returns the position of the pivot
```

The quicksort algorithm recursively sorts the arrays on either side of the position returned by PARTITION. The running time of this algorithm depends on the choice of the pivot. If the element in position `high` partitions the array A into almost similar-sized sub-arrays, then quicksort finishes the sorting quickly. On the other hand, if the partition induced by `high` is skewed, then the algorithm will perform poorly. The running time of the algorithm is proportional to the number of comparisons performed by PARTITION. Let $T(n)$ denote the number of comparisons performed by QUICKSORT on an array of size n. We can express $T(n)$ as

...with base case $T(0) = 1$.

$$T(n) \leq \max_{i \in \{1, n-1\}} \{T(i-1) + T(n-i) + O(n)\}.$$

Solving this gives $T(n) = O(n^2)$. Indeed, if the array is sorted in the decreasing order, then the running will be $\Theta(n^2)$. Ideally, we would like a pivot that divides the array into almost equal-sized sub-arrays. This would give a recurrence relation of the form

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$$

which solves to $T(n) = \Theta(n \log n)$.

To avoid this type of an adversarial input that can skew the running time of the algorithm, we can think of an alternative *randomized* solution - choose the pivot uniformly at random from the set of all indices. The pseudocode is given as Algorithm 2.

Algorithm 2.2: RANDOMIZEDQUICKSORT$(A, \texttt{low}, \texttt{high})$

```
if low < high then
    //Choose a random index between low and high
    pos ← random(low, high)
    Swap $A[\texttt{pos}]$ and $A[\texttt{high}]$
    pivot ← PARTITION$(A, \texttt{low}, \texttt{high})$
    RANDOMIZEDQUICKSORT$(A, \texttt{low}, \texttt{pivot}-1)$
    RANDOMIZEDQUICKSORT$(A, \texttt{pivot}+1, \texttt{high})$
```

With these random choices, the running-time of the algorithm is no longer fixed. It could vary from $\Theta(n \log n)$ if the random choices that the algorithm made were all very good, to $\Theta(n^2)$ if the random choices all gave skewed partitions. Thus the running time of the algorithm is a random variable whose distribution is determined by the random choices of the algorithm. Analyzing the algorithm amounts to analyzing this random variable under the worst-case input.

Remember that the random choices are inherent to the algorithm. We do not make any assumption on the distribution that generates the input. Thus the analysis of a randomized algorithm is a worst-case analysis. We will argue that *for every input* the *expected running time* is small. The expectation or average here is on the random choices of the algorithm, **not on any assumption on the input**. For the randomized version of quicksort, we will denote by $T(n)$, the expected running time of the algorithm under worst-case inputs.

To understand the intuition behind why a random choice of the pivot can make the algorithm perform better, let us understand when the choice of the pivot is good. Let $A = (a_1, a_2, \dots, a_n)$ be the input array, and let $B = (b_1, b_2, \dots, b_n)$ denote the sorted version of A. Suppose that the pivot chosen by the algorithm partitions the array A into two parts each such that the smallest part has size at least $n/10$, and the largest has size at most $9n/10$. If this holds, then the recurrence of the running time becomes

$$T(n) \leq T(\alpha n) + T((1-\alpha)n) + O(n),$$

for some constant α.

Solving this gives $T(n) = \Theta(n \log n)$. If we choose a pivot at random, then the probability that the pivot partitions A in this manner is $8/10$. Thus, with 80% probability we are likely to choose a pivot that partitions the array into two parts that are almost equal. We will look at two analyses of the algorithm.

2.3.1 A direct analysis

Since the pivot is chosen uniformly at random, the probability that the i^{th} smallest element in the array is chosen is $1/n$. If the pivot chosen is the i^{th} smallest, then the subproblems have size n and $n-i$.

The running time $T(n)$ is a random variable, and we are interested in the quantity $\mathbb{E}[T(n)]$. To write this expectation, let p_i denote the probability that the random choice of the pivot returns the i^{th} smallest element in the array and let E_i denote the corresponding event. Then we have

$$\begin{aligned}\mathbb{E}[T(n)] = \sum_{i=1}^{n} p_i \mathbb{E}[T(n) \mid E_i] &\leq \sum_{i=1}^{n} \frac{1}{n} \mathbb{E}[\Theta(n) + T(i) + T(n-i)] \\ &\leq \Theta(n) + \frac{1}{n} \sum_{i=1}^{n} (\mathbb{E}[T(i)] + \mathbb{E}[T(n-i)])\end{aligned}$$

Thus, we can write the expected running time of randomized quicksort as

$$\mathbb{E}[T(n)] = \Theta(n) + \frac{2}{n}\sum_{i=0}^{n-1}\mathbb{E}[T(i)] \le kn + \frac{2}{n}\sum_{i=1}^{n-1}\mathbb{E}[T(i)].$$

This recurrence is a little unwieldy to solve directly, and we need a good guess to use the substitution method. We will go back to our intuition that with probability $4/5$, the randomly chosen pivot partitions the array nicely. With this in mind, let us guess that $\mathbb{E}[T(i)] \le ci\log i$ for a suitable constant c we will fix later. With this guess, we have

$$\mathbb{E}[T(n)] \le kn + \frac{2}{n}\sum_{i=1}^{n-1} ci\log i,$$

and we need to verify that $T(n) \le cn\log n$. To that end, we first look at the following sum

$$\begin{aligned}\sum_{i=1}^{n-1} i\log i &\le \sum_{i=1}^{n/2} i\log i + \sum_{n/2}^{n} i\log i\\ &\le \log\left(\frac{n}{2}\right)\sum_{i=1}^{n/2} i + \log n\left(\sum_{i=1}^{n} i - \sum_{i=1}^{n/2-1} i\right)\\ &\le \frac{n^2}{8}\log\left(\frac{n}{2}\right) + \frac{3n^2}{8}\log n\\ &\le \frac{n^2}{2}\log n - \frac{n^2}{8}.\end{aligned}$$

Substituting this in the recurrence, we have

$$\begin{aligned}\mathbb{E}[T(n)] &\le kn + \frac{2}{n}\left(\frac{cn^2}{2}\log n - \frac{cn^2}{8}\right)\\ &\le kn + cn\log n - \frac{cn}{4}\\ &\le cn\log n, \text{ when } c > 4k.\end{aligned}$$

2.3.2 A tighter bound

We will now see a different analysis of the running time of quicksort that uses the properties of random variables that we saw earlier. Recall that the number of comparisons performed by quicksort (which is proportional to the running time of the algorithm) is a random variable - let's denote it by X. We are interested in computing $\mathbb{E}[X]$.

Instead of analyzing X directly, we will look at indicator random variables X_{ij} (for $i < j$) that is set to 1 if b_i and b_j are compared at any point in the run of randomized quicksort. We can express the random variable X as follows:

$$X = \sum_{i<j} X_{ij}.$$

Using linearity of expectation, we can express $\mathbb{E}[X]$ as

$$\mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}].$$

We will now look at bounding the expectation of X_{ij}. Consider the elements $b_i < b_{i+1} < \cdots < b_j$, and consider the first time in the execution of quicksort when a pivot p is chosen from the interval $[i, j]$. If $i < p < j$, then PARTITION will divide the array into two parts and b_i and b_j will be in different parts. After this b_i and b_j are never compared, and hence $X_{ij} = 0$. Thus, the only way that $X_{ij} = 1$ is when either i or j is chosen as the pivot the first time an element in the interval $[i, j]$ is chosen as pivot. Since the pivot is chosen uniformly at random, we have

$$\mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = \frac{2}{j-i+1}.$$

Plugging this in the expression for $\mathbb{E}[X]$, we get

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i<j} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&= \sum_{k=2}^{n} (n-k+1) \frac{2}{k} \\
&= 2(n+1) \sum_{k=2}^{n} \frac{1}{k} - 2(n-1) = 2(n+1) \sum_{k=1}^{n} \frac{1}{k} - 4n
\end{aligned}$$

The first term in the RHS is the harmonic sum and is equal to $\ln n + \Theta(1)$. This gives the value of the expectation as

$$\mathbb{E}[X] = 2n \ln n + \Theta(n).$$

3 Dictionaries

One of the most basic primitives in any algorithm or computation is to store data and look it up later. The Dictionary is one of the most basic abstract data types (ADT) used in many algorithms. We will see multiple data structures that implement the Dictionary.

A dictionary consists of a set of $(\mathsf{key}, \mathsf{value})$ pairs with the condition that each key appears at most once in the dictionary. The keys are members of a totally ordered set, and hence can be sorted. The ADT Dictionary supports the following operations:

- $\mathsf{Insert}(k, v)$ - insert the pair (k, v) in the dictionary. We will assume that the inserts all come for different key values. It is possible to do a search first to make sure that the key does not exist in the dictionary.

- $\mathsf{Delete}(k)$ - delete the pair with the key k from the dictionary.

- $\mathsf{Search}(k)$ - search of the pair with key k in the dictionary.

We will look at data structures that can support all the operations of a Dictionary as efficiently as possible. We will see how randomization can be used to give implementations that perform well on the average (over the randomness of the algorithm) for worst-case inputs. We will see binary search trees with good amortized bounds as well.

3.1 Hash tables

A hash table consists of an array Arr of size t that is indexed using the key values. We have a function hash such that $\mathsf{hash}(k)$ returns a number between 1 and t. The key is then inserted into the location $\mathsf{Arr}[\mathsf{hash}(k)]$. Typically the universe $\mathcal{U}$ from which the keys are obtained is much larger than the size of the array t. Thus, irrespective of the choice of the hash function, there will exist at least (in fact, many) two keys k_1 and k_2 such that $\mathsf{hash}(k_1) = \mathsf{hash}(k_2)$. This is known as a *collision*. A good design should choose the hash function carefully, and take care of the collisions in an effective manner.

We will look at how to take care of collisions, and the choice of hash functions. We start with a simple method of handling collisions, known as *chaining*.

3.1.1 Hashing with chaining

In hashing with chaining, we have an array Arr indexed from 1 to t, where each entry in the array points to a linked list. The linked list corresponding to Arr$[i]$ contains the keys that are hashed to i. Let $h : \mathscr{U} \to [t]$ be the hash function we have chosen. The operations of the Dictionary ADT are implemented as follows:

- Insert(k, v): Compute $h(k)$. Insert (k, v) into the linked list pointed to from Arr$[h(k)]$.
- Delete(k): Compute $h(k)$. Scan the linked list pointed to from Arr$[h(k)]$ and delete the node with the key value k.
- Search(k): Compute $h(k)$. Scan the linked list pointed to from Arr$[h(k)]$. If k is found, return the corresponding value v.

Notice that the running time of all three operations depend on the length of the linked list. The length of the list depends on the choice of the hash function.

We would like all the operations to be $O(1)$, but this would be impossible to achieve in the worst-case. We will settle for something weaker, but practically very good.

Typically the universe $\mathscr{U}$ of keys is such that that $|\mathscr{U}| \gg t$, and hence irrespective of the choice of the hash functions, there will be multiple keys in the universe that map to the same hash value. Even though $|\mathscr{U}|$ is large, the actual subset of the universe that will be hashed is a number $n = \Theta(t)$. The only problem is that we do not know the n elements in the universe that will be hashed.

If we were to choose the hash function $h : \mathscr{U} \to [t]$ beforehand, it is always possible to adversarially choose n elements such that there are a large number of collisions. If the choice of the n elements are adversarial, we will need randomness in the choice of the hash function to achieve good bounds on the running time of inserts, searches, and deletes.

One ideal scenario is that the hash function h is chosen uniformly at random from the set of all functions from $\mathscr{U}$ to $[t]$. This is not practical, since there are $|t|^{|\mathscr{U}|}$ many functions, and storing any such function will require $|\mathscr{U}| \log t$ space. Instead, we could have just maintained an array of size $|\mathscr{U}|$, and used the trivial hash function $h(x) = x$. This would avoid all collisions, at the cost of using space proportional to the size of the universe $|\mathscr{U}|$.

Consider the case where the universe $\mathscr{U}$ is the set of all IP addresses. Here the universe has size 2^{128}, whereas a data structure that is used to store the IPs that access a server is considerable smaller than this number.

Nonetheless, let us consider the case of a random hash function as a warm-up. To set it up formally, we have a set $\mathscr{H} = \{h : \mathscr{U} \to [t]\}$ (the set of all functions). A random hash function is obtained by choosing a function uniformly at random from $\mathscr{H}$. Thus, the probability that a particular function h is obtained is $1/|\mathscr{H}|$.

Another way to view is that for each $x \in \mathscr{U}$, the value of $h(x)$ is chosen uniformly at random from the set $[t]$. Thus, for a random hash function we can say the following.

$$\Pr_{h \in_r \mathscr{H}} [h(x) = i] = \frac{1}{t}.$$

Suppose we have a set S with n elements that we want to hash into the table of size t, then we can bound the search time for any $x \in \mathscr{U}$ as follows.

Since the hash function is random, the search time is a random variable, and we will look at the *expected cost* of the search operation. Observe that the insert and delete operations take asymptotically the same time.

Let $x \in \mathscr{U}$ be any elements. For every $y \in S$, the probability that $h(x) = h(y)$ can be bounded as follows.

$$\Pr_{h \in_r \mathscr{H}}[h(x) = h(y)] = \sum_{i \in [t]} \Pr[h(x) = i \wedge h(y) = i]$$

Since h is chosen uniformly at random, the events $h(x) = i$ and $h(y) = i$ are independent. Thus, we can write

$$\Pr_{h \in_r \mathscr{H}}[h(x) = h(y)] = \sum_{i \in [t]} \Pr[h(x) = i] \cdot \Pr[h(y) = i] = \sum_{i \in [t]} \frac{1}{t^2} = \frac{1}{t}.$$

The expected search time depends on the number of elements $y \in S$, that collide with x in the hash table - i.e. $h(x) = h(y)$. To calculate this quantity, define the following indicator random variables.

$$I_y = \begin{cases} 1 & \text{if } h(x) = h(y) \\ 0 & \text{otherwise} \end{cases}$$

Now, the search time for the element $x \in \mathscr{U}$ is the random variable $I = \sum_{y \in S} I_y$. Using the linearity of expectations, we can say that $\mathbb{E}[I] = |S|/t$. Thus, when $|S| = \Theta(t)$, we have the expected search time as $\Theta(1)$.

Even though the expected behaviour is good, it can be shown that even using random hash functions, the worst case bound on the running time can be $\Theta(\log n / \log\log n)$ when $n = \Theta(t)$.

Balls into bins and the birthday problem

Before we go into discussing about hash families that are not fully random, we will look at how large the hash table should be so that the number of collisions for any element is only $O(1)$ in the worst-case when a purely random hash function is used. The discussion in the last section showed that is $t = O(|S|)$, the expected search time is $\Theta(1)$.

For a set $S \subseteq \mathscr{U}$, a purely random hash function h mapping S to $[t]$ can be thought as a $m = |S|$ balls being thrown uniformly, and independently at random into t bins. We are interested computing the probability that no bin has more than one ball in it. In other words, every position of the hash table has at most one element of the universe hashed into it.

Let E_i be the event that the i^{th} element in S did not cause a collision given that the first $i-1$ elements did not cause collision. We are interested in giving an upper bound on the event $E = \overline{E}_1 \cup \overline{E}_2 \cup \cdots \cup \overline{E}_m$. The event E covers the case that there is at least one collision when hashing m elements into a hash table of size t. By the union bound that we saw earlier, we can write the probability for E as

$$\Pr[E] \leq \sum_{i=1}^{m} \Pr[\overline{E}_i].$$

Since the first $i-1$ elements did not cause collision, they all hashed into different position on the table. For i to not cause a collision, i should be hashed to a position different from these $i-1$ positions. Since there are t positions on the hash table, this probability can be expressed as

$$\Pr[\overline{E}_i] = \frac{i-1}{t}.$$

Putting all this together, we have

$$\begin{aligned}\Pr[E] &\leq \sum_{i=1}^{m} \frac{i-1}{t} = \frac{1}{t}\sum_{i=1}^{m-1} i \\ &= \frac{m(m-1)}{2t} \leq \frac{m^2}{2t}\end{aligned}$$

If $t = m^2$, the $\Pr[E]$ is small ($< 1/2$). Hence, w.h.p there are no collisions and the worst-case search time is $O(1)$.

This bound of $t = \Theta(m^2)$ is in fact tight in the sense that if t is smaller than m^2, then with probability at least $1/2$, two elements in S will hash into the same position in the table. Let's see why this is the case. As earlier, we will look at the event E_i that the i^{th} element does not cause a collision given that the first $i-1$ elements did not cause a collision. Thus, the probability of the event E' that every elements is hashed into a separate position in the hash table is bounded by

This is known as the birthday problem.

$$\Pr[E'] = \left(1-\frac{1}{t}\right)\left(1-\frac{2}{t}\right)\cdots\left(1-\frac{m-1}{t}\right)$$

Using the bound that $1-x \leq e^{-x}$, we can upper bound this probability as

$$\begin{aligned}\Pr[E'] &\leq e^{-1/t} \cdot e^{-2/t} \cdots e^{-(m-1)/t} \\ &= e^{-\frac{1+2+\cdots+(m-1)}{t}} = e^{-\frac{m(m-1)}{2t}}.\end{aligned}$$

Thus, if $t \leq m^2/4$, then $\Pr[E'] \leq e^{-2} < 1/2$. Hence for $t < m^2/4$, with probability at least $1/2$, there are at least two elements that hash into the same position on the hash table.

3.1.2 Universal hash families

While random hash functions behave well in expectation, we saw how they are impractical. We need to trade-off on true randomness for practicality, yet try to achieve similar bounds on the running time for Insert, Search, and Delete. One of the requirements for good bounds on the running time of these operations is what is defined using the notion of *universal hash families*.[1]

[1] J Lawrence Carter and Mark N Wegman. "Universal classes of hash functions". In: *Journal of Computer and System Sciences* 18.2 (1979), pp. 143–154.

Definition 3.1 (Universal hash families). *A set $\mathcal{H}$ of functions from $\mathcal{U}$ to $[t]$ is said to a* universal hash family *if for every $x \neq y \in \mathcal{U}$,*

$$\Pr_{h \in_r \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{t}$$

A few remarks are in order here. Note that the set of all functions from $\mathscr{U}$ to $[t]$ is universal. The universality follows from the property that for a random function sampled from the set of all function, the image for any element $x \in \mathscr{U}$ is equally likely to be any number in $[t]$. This may not be true for every universal hash family. Uniformity on its own is also not very useful for hashing. Consider the family $\mathscr{T}$ to be the set of all constant functions - i.e. $\mathscr{T} = \{h_i \mid i \in [t]\}$, where $h_i(x) = i$ for every $x \in \mathscr{U}$. You can see that for every x and i

There is a notion of strong universality that also implies the uniformity property. Many of the constructions of universal hash families also give strong universility.

$$\Pr_{h \in_r \mathscr{H}} [h(x) = i] = \frac{1}{t}.$$

But, if h is a hash function sampled from $\mathscr{T}$, then for every $x, y \in \mathscr{U}$, $h(x) = h(y)$.

Exercise 3.1. Verify that the expected running-time for Search, Insert, and Delete is $\Theta(1)$ when the hash function is sampled from a universal hash family.

3.1.3 Multiplicative hashing

We will now see a family of universal hash functions that can be described succinctly, and computed efficiently. This family requires choosing a large prime number, larger than the size of the universe $\mathscr{U}$. We will then see a slight variant of this idea that avoids the need of a large prime number but only gives *near-universality*. It will not be hard to verify that this notion of near-universality (that we will define later) is sufficient for the running-time bounds that we need.

3.1.3.1 A universal family

Let p be a prime number such that $p > |\mathscr{U}|$. The family of hash functions is defined using two parameters, $a \in \{1, \ldots, p-1\}$ and $b \in \{0, 1, \ldots, p-1\}$ as follows:

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod t$$

The number of hash functions in this family is at most p^2, and choosing a hash function requires choosing a and b uniformly at random from the set $\{1, \ldots, p-1\}$ and $\{0, 1, \ldots, p-1\}$, respectively. Since modular arithmetic can be done efficiently, computing $h(x)$ for any x is also efficiently doable. To show that the family is universal, we need to prove the following statement.

Theorem 3.2. *For any $x \neq y \in \mathscr{U}$,*

$$\Pr_{a,b}\left[h_{a,b}(x) = h_{a,b}(y)\right] \leq \frac{1}{t}.$$

We will the following lemma about prime numbers to prove the theorem.

Lemma 3.3. *Let p be a prime number. For every $a \in \{1,\dots,p-1\}$, there exists a unique $x \in \{1,\dots,p-1\}$ such that $ax \equiv 1 (\text{mod } p)$.*

Proof. First, we will show that for $a \in \{1,2,\dots,p-1\}$, there does not exist an $x \in \{1,2,\dots,p-1\}$ such that $ax \equiv 0 (\text{mod } p)$. This is because if p divides ax, then p must divide either a or x since p is a prime. But, this is not possible since $a, x < p$.

Now, we will show that for two different values $x \neq x' \in \{1,2,\dots,p-1\}$, we cannot have $ax \equiv ax' (\text{mod } p)$. Together with the first statement we proved, this means that the set $\{ax (\text{mod } p) \mid x \in \{1,2,\dots,p-1\}\}$ has $p-1$ elements, and hence there is a unique x such that $ax \equiv 1 (\text{mod } p)$.

Assume that $ax \equiv ax' (\text{mod } p)$. Then, we have $a(x-x') \equiv 0 (\text{mod } p)$. Once again this means that p must divide a or $x-x'$. Since $a < p$, it is impossible that p divides a. Since $1 \le x, x' \le p-1$, $|x-x'| < p-2$ and hence p does not divide $x-x'$. Thus, it must be the case that $x = x'$. □

Proof of Theorem 3.2. Fix $x, y, r, s \in \{0,2,\dots,p-1\}$ where $x \neq y$. Consider the system of equations

$$ax + b \equiv r (\text{mod } p)$$
$$ay + b \equiv s (\text{mod } p).$$

Using Lemma 3.3, we can say that this system has a unique solution for a and b, given by $a = (r-s)(x-y)^{-1}$ and $b = (ry - xs)(y-x)^{-1}$. Thus, we can say that

$$\Pr_{a,b}\left[(ax+b) \bmod p = r \wedge (ay+b) \bmod p = s\,\right] = \frac{1}{p(p-1)}.$$

Now, we can say that

$$\Pr_{a,b}\left[h_{a,b}(x) = h_{a,b}(y)\right] = \frac{N}{p(p-1)},$$

where N is number of pairs (r,s) such that $r \neq s$ and $r \equiv s (\text{mod } p)$. To bound N, note that for every $r \in \{0,1,\dots,p-1\}$, there are at most $\lfloor \frac{p}{t} \rfloor$ values of $s \neq r$ such that $s \equiv r (\text{mod } p)$. Since p is prime $\lfloor \frac{p}{t} \rfloor$ is at most $\frac{p-1}{t}$. Therefore $N \le p(p-1)/t$, and hence

$$\Pr_{a,b}\left[h_{a,b}(x) = h_{a,b}(y)\right] = \frac{1}{t},$$

□

3.1.3.2 A near-universal family

We will now look at a simpler family of multiplicative hashing where we are not required to choose a large prime. Instead of a universal family, we will obtain near-universality which is defined as follows.

Definition 3.4 (Near-universal hash family). *A family $\mathscr{H}$ of hash functions* *from $\mathscr{U}$ to $[t]$ is said to be near-universal if for every $x \neq y \in \mathscr{U}$,*

$$\Pr_{h \in_r \mathscr{H}} [h(x) = h(y)] \leq \frac{2}{t}.$$

EXERCISE 3.2. Verify that near-universality is sufficient to obtain $\Theta(1)$ expected time for Insert, Search, and Delete.

For this construction, we will assume that $|\mathscr{U}| = 2^w$ for some w and $t = 2^\ell$ for some ℓ. The family of hash functions is defined by choosing an odd number $a \in \{0, 1, \ldots, 2^w - 1\}$ and having

$$h_a(x) = \left\lfloor \frac{ax (\text{mod } 2^w)}{2^{w-\ell}} \right\rfloor$$

After sampling a random odd integer $a \in \{0, 1, \ldots, 2^w - 1\}$, computing $h_a(x)$ is quite easy to implement in most modern programming languages. If w is the number of bits of integer type, then $ax(\text{mod } 2^w)$ can be computed by a multiplication operation. Now, the division by $2^{w-\ell}$ amounts to a right shift by $w - \ell$ bits.

We will prove the following statement to show that the hash family is near-universal.

THEOREM 3.5. *Let W denote the set of odd integers in $\{0, 1, \ldots, 2^w - 1\}$. For any $x \neq y \in \{0, 1, \ldots, 2^w - 1\}$,*

$$\Pr_{a \in_r W} [h_a(x) = h_a(y)] \leq \frac{2}{2^\ell}.$$

Before we start the proof let's try to understand the function $h_a(x)$. Since a and x are w-bit numbers, ax is a $2w$-bit number. Thus, $ax(\text{mod } 2^w)$ gives the last (least significant) w bits of the product ax. Now, dividing this by $2^{w-\ell}$ gives the first ℓ bits of these w bits.

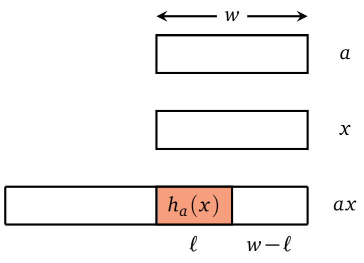


FIGURE 3.1: Computing $h_a(x)$ for an odd $a \in \{0, 1, \ldots, 2^w - 1\}$.

If $x \neq y$ are such that $h_a(x) = h_a(y)$, then clearly these ℓ bits are identical. Let's assume (w.l.o.g) that $x < y$. Now, depending on whether $ax(\text{mod } 2^w) \leq ay(\text{mod } 2^w)$ or not, the first ℓ bits of the least significant w bits of $a(x-y)(\text{mod } 2^w)$ are either 1^ℓ or 0^ℓ, respectively. Therefore, if $h_a(x) = h_a(y)$, then $h_a(x-y) = 0$ or $h_a(x-y) = 2^\ell - 1 = t - 1$. Hence we have

$$\Pr_{a \in_r W} [h_a(x) = h_a(y)] \leq \Pr_{a \in W} [h_a(x-y) = 0] + \Pr_{a \in_r W} [h_a(x-y) = t-1].$$

To proceed further and analyze the R.H.S of this equation, we need the following lemma.

LEMMA 3.6. *Let $x, z \in W$ be any two integers. There exists exactly one $a \in W$ such that $ax(\text{mod } 2^w) = z$.*

Proof. Suppose that there exists a, a' such that $ax(\text{mod } 2^w) = a'x(\text{mod } 2^w)$. Then we have $(a - a')x \equiv 0(\text{mod } 2^w)$. Since x is odd, 2^w must divide $a - a'$.

But, $a, a' \in W$ and hence $|a - a'| < 2^w$. Thus, it must be the case that $a = a'$. Thus, there is at most one a such that $ax (\text{mod } 2^w) = z$, and the function $f_x(a) = ax (\text{mod } 2^w)$ is injective. Since $f_x : W \to W$ is injective, it must be a bijection as well. Hence for every $z \in W$, there is exactly one $a \in W$ such that $ax (\text{mod } 2^w) = z$. □

One consequence of the lemma is that for any $z, x \in W$,

$$\Pr_{a \in_r W} [ax (\text{mod } 2^w) = z] = \frac{1}{2^{w-1}}.$$

In other words, for a random choice of a, the number $ax (\text{mod } 2^w)$ is uniformly distributed over W. With this in hand, we will complete the proof of the near-universality.

Proof of Theorem 3.5. Let $x \neq y$ and assume (w.l.o.g) that $x < y$. Thus, $x - y (\text{mod } 2^w) = q2^r$, where q is an odd integer. From Lemma 3.6, we know that $\Pr_{a \in_r W} [aq (\text{mod } 2^w) = z] = 1/2^{w-1}$ for every $z \in W$. Thus, we can think of $aq (\text{mod } 2^w)$ as a w-bit number whose last bit is 1 and the first $w - 1$ bits are chosen uniformly at random. Hence $aq2^r (\text{mod } 2^w)$ (which is equivalent to $aq (\text{mod } 2^w) \cdot 2^r (\text{mod } 2^w)$) has the last r bits all 0, the next bit a 1, and the remaining $w - r - 1$ bits chosen uniformly at random. Now, when we look at $h_a(x - y)$, we need to look at the first ℓ bits of this w-bit number. We will look at different cases depending on the relative sizes of r and ℓ.

- $r > w - \ell$: In this case, the number ends with a 1 followed by a non-zero number of 0s. Hence

$$\Pr_{a \in W} [h_a(x - y) = 0] = \Pr_{a \in_r W} [h_a(x - y) = t - 1] = 0.$$

- $r < w - \ell$: In this case $w - \ell \geq r + 1$, and hence $h_a(x - y)$ consists of ℓ uniformly random bits. Therefore

$$\Pr_{a \in W} [h_a(x - y) = 0] = \Pr_{a \in_r W} [h_a(x - y) = t - 1] = \frac{1}{2^\ell} = \frac{1}{t}.$$

- $r = w - \ell$: In this case, the number $h_a(x - y)$ consists of $\ell - 1$ random bits followed by a 1. Hence

$$\Pr_{a \in W} [h_a(x - y) = 0] = 0$$
$$\Pr_{a \in_r W} [h_a(x - y) = t - 1] = \frac{1}{2^{\ell - 1}} = \frac{2}{2^\ell} = \frac{2}{t}.$$

Thus in all cases, we have

$$\Pr_{a \in W} [h_a(x - y) = 0] + \Pr_{a \in_r W} [h_a(x - y) = t - 1] \leq \frac{2}{t}.$$

□

3.1.4 Perfect hashing

Consider the following situation that is referred to as the static dictionary problem - you are given a set $S \subseteq \mathscr{U}$ that will remain fixed throughout all the search queries, given an $x \in \mathscr{U}$ check if $x \in S$? Since the set S is fixed before the start of the search queries, you are allowed time to preprocess and find the best way to store S. Ideally, we want to save S in such a way that all queries of the form "Is $x \in S$?" can be answered in $O(1)$ worst-case time. This is known as the *perfect hashing problem*. The constructions given earlier will not work since even if assume random hash functions, we have $\omega(1)$ collisions in the worst-case.

The method that we will see now, known as FKS hashing, that gives a way to store S in $O(|S|)$ space and provides $O(1)$ worst-case time for searches. We saw that $O(1)$ search-time is unattainable using the simple hashing that we described earlier. The idea of Fredman, Komlos, and Szemeredi was to use two level of hash tables - one for the primary table, and a second level of hash tables for each of the items that collide in the primary hash table. Let us see how to put this into practice. We will assume that we have a universal/near-universal hash family $\mathscr{H}$ at our disposal - any one from the previous sections will do.

FKS are the initials of Fredman, Komlos, and Szemeredi who described this method in 1984.

We will start with two statements about the number of collisions in a hash table when a hash function from a universal hash family is used for hashing.

Lemma 3.7. *Suppose that we hash a set S of size n into a hash table of size t using a universal hash family $\mathscr{H}$, then with probability at least $1/2$, the total number of collisions across all the positions in the table is at most $O(n^2/t)$.*

Proof. For two elements $i \neq j$ in S, we have

$$\Pr_{h \in_r \mathscr{H}}[h(i) = h(j)] \leq \frac{1}{t}.$$

Let X_{ij} denote the indicator random variable that 1 if i and j collide when h is sampled uniformly at random from $\mathscr{H}$. Thus the total number of collisions is given by the random variable $X = \sum_{i \neq j} X_{ij}$. Therefore we have

$$\mathbb{E}[X] = \sum_{i \neq j} \mathbb{E}[X_{ij}] \leq \binom{n}{2}\frac{1}{t} \leq \frac{n^2}{2t}.$$

Since X is a random variable that takes non-negative integral values, we can write

$$\mathbb{E}[X] = \sum_{i \geq 0} \Pr[X \geq i] \geq \sum_{i=0}^{n^2/t} \Pr[X \geq i] \geq \frac{n^2}{t} \Pr[X \geq n^2/t].$$

Thus, we have $\Pr[X \geq n^2/t] \leq 1/2$. $\square$

Consequently, if $t = n$, then the number of collisions is atmost n (w.p $1/2$) and if $t = n^2$, then the number of collisions is at most 1 (w.p $1/2$).

The idea of FKS hashing is as follows. We first sample h from $\mathscr{H}$ uniformly at random and count the number of collisions, discarding h and resampling if the total number of collisions while hashing the set S is more than n. Since the probability of finding an h with collisions at most n is at least $1/2$, we can find such an h in $O(1)$-time w.h.p. For each $i \in [t]$, let n_i be the number of elements $x \in S$ such that $h(x) = i$. From the choice of h we can say that

$$\sum_{i=1}^{t} \binom{n_i}{2} \leq n.$$

Now, for each i, we choose hash function h_i from a universal hash family that maps these n_i elements into a hash table of size n_i^2. From the earlier calculation, we know that with probability at least $1/2$, a random hash function from a universal family will satisfy this property. Thus, for each i, we can find such a hash function in time $O(1)$. We use h as the primary hash function, and use h_i as the secondary hash function to take care of collisions in the primary table. The collisions in h_is can be taken care of using linked lists. Once the hash functions are chosen and S stored, we can answer each query of "Is $x \in S$?" in $O(1)$ worst-case time. Furthermore, the total space taken (outside the space for storing the hash functions) is at most

$$n + \sum_{i=1}^{t} n_i^2 \leq n + 4\sum_{i=1}^{t} \binom{n_i}{2} = \Theta(n).$$

3.1.5 Open addressing

Open addressing is another hashing technique to take care of collisions that might occur due to the choice of the hash functions. In open addressing, the collisions are taken care of by inserting the element x such that $\mathsf{Arr}[h(x)]$ is occuppied to another empty location in the hash table. This empty location is computed in different ways. We will concentrate on one such method known as *linear probing*.

Open addressing with linear probing works well in practice due to locality of reference. On a cache miss a contiguous portion of the array will get loaded on to the cache, and the subsequent searches via linear probing can be done without any further cache misses. Knuth[2] was the first to show that open addressing with linear programming takes $O(1)$-time in expectation if the hash function is truly random. His article title "Notes on "Open" addressing" says that it is his first analysis of an algorithm. We will try to obtain theoretical bounds on the search time when we use linear probing.

[2] Donald E Knuth. "Notes on "open" addressing". In: *Unpublished memorandum* (1963), pp. 11–97. URL: https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf.

For the remainder of this subsection, we will assume that the hash function is truly random. While the assumption is unrealistic, the analysis will become easier. Weaker assumptions based on universality are sufficient to obtain these bounds, but that analysis is more sophisticated.

First, let us see how to implement the ADT operations using linear probing. Each entry of the hash table Arr consists of one of three possible items.

- A value x which is the element that is stored in the hash table at that location.
- A value $\perp$ that denotes that the location in the hash table is empty.

- A value $\top$ that denotes a *tombstone* - a location where a value was present, but which was deleted at some point in the past.

The size of the hash table will be such that at least half the positions will be $\perp$. Thus, we maintain an additional counter of the number of elements present in the table as well as the number of tombstone $\top$. Once the number of $\perp$ drops below half the total table size, the entire contents of the hash tbale (except the tombstones) are rehashed into a new table. The ADT operations are performed as follows.

- $\mathsf{Insert}(k,v)$: Starting from $h(k)$, insert (k,v) into the first location i after $h(k)$ such that $\mathsf{Arr}[i] = \perp$ or $\mathsf{Arr}[i] = \top$.

- $\mathsf{Search}(k)$: Starting from $h(k)$ continue searching for k until position i such that $\mathsf{Arr}[i] = \perp$.

- $\mathsf{Delete}(k)$: Starting from $h(k)$ continue until k is found in some position i, in which case set $\mathsf{Arr}[i] = \top$. If we encounter $\perp$ before finding k, then k is not present in the hash table.

The running time of all three operations depend on the time for searching a key k in the hash table. The cost of rehashing will not be considered for now. Even though the cost of one rehashing could be as high as $O(n)$ if there are n elements present in the table, this is not performed often and hence the *amortized* cost per operation is small. We will look at amortized analysis at a later stage, and for now assume that the total number of elements inserted into the hash table across its entire history is less than half of the table size t.

To analyze the running time, we will define the notion of a *run*. A run is a maximal contiguous sequence of positions in the hash table that are all occupied by elements of $\mathcal{U}$ or by tombstones $\top$. If we are searching for $x \in \mathcal{U}$ and $h(x)$ is part of a run of length k, then the running time for the search could be $O(k)$. The following lemma shows that if the hash table size is large enough, then there are unlikely to be long runs.

Lemma 3.8. *Let n be the total number of elements that are inserted into a hash table of size $t = 2n$. For any i, the probability that there is a run of lenght k starting at i is at most c^k for $c < 1$.*

Proof. If there is a run of length k starting from i, then exactly k elements must be mapped via the hash function into the positions $i, i+1, \ldots, i+k-1$. So, we will bound this probability. Let $p_{i,k}$ denote this probability. We can then write

$$\begin{aligned} p_{i,k} &= \binom{n}{k}\left(\frac{k}{t}\right)^k\left(1-\frac{k}{t}\right)^{n-k} \\ &= \frac{n!}{k!(n-k)!}\left(\frac{k}{t}\right)^k\left(\frac{t-k}{t}\right)^{n-k} \end{aligned}$$

Using Stirling's approximation that $n! \approx \sqrt{2\pi n}(n/e)^n$, we can write the R.H.S as follows.

$$p_{i,k} \leq \frac{1}{\sqrt{2\pi}}\sqrt{\frac{n}{k(n-k)}}\frac{n^n}{k^k(n-k)^{n-k}}\frac{k^k}{t^n}(t-k)^{n-k}$$

Simplifying this further assuming that $t = 2n$, we get that

$$\begin{aligned}
p_{i,k} &\leq \frac{1}{\sqrt{2\pi}}\sqrt{\frac{n}{k(n-k)}}\frac{1}{2^n}\left(\frac{2n-k}{n-k}\right)^{n-k} \\
&= \frac{1}{\sqrt{2\pi}}\sqrt{\frac{n}{k(n-k)}}\frac{1}{2^k}\left(\frac{2n-k}{2(n-k)}\right)^{n-k} \\
&= \frac{1}{\sqrt{2\pi}}\sqrt{\frac{n}{k(n-k)}}\frac{1}{2^k}\left(1+\frac{k}{2(n-k)}\right)^{n-k}
\end{aligned}$$

Using the bound that $1 + x \leq e^x$, we can simplify this as follows.

$$p_{i,k} \leq \frac{1}{\sqrt{2\pi}}\sqrt{\frac{n}{k(n-k)}}\left(\frac{\sqrt{e}}{2}\right)^k \leq c^k, \text{ for some constant } c < 1$$

□

We can now bound the running time for a search operation for a key x. Let $i = h(x)$ be the position on the hash table mapped by h for x. If there is a run containing i of length k, then the cost of the search operations is $O(k)$. We will bound the expected value of k.

LEMMA 3.9. *Let $x \in \mathcal{U}$ be any element in the universe. Let r_x be the length of the run containing $h(x)$ in the hash table of size t. Then $\mathbb{E}[r_x] = O(1)$.*

Proof. We can write the expectation as follows.

$$\mathbb{E}[r_x] = \sum_{\ell=0}^{t} \ell \cdot \Pr[r_x = \ell]$$

Suppose that $h(x) = i$. Then if $r_x = \ell$, there is a position between $i - \ell + 1$ to i such that there is a run of length ℓ that starts from that position. Thus, we can rewrite the equation as follows.

$$\begin{aligned}
\mathbb{E}[r_x] &\leq \sum_{\ell=0}^{t} \ell \left(\sum_{j=i-\ell+1}^{i} p_{j,\ell}\right) \\
&\leq \sum_{\ell=0}^{t} \ell^2 c^\ell = O(1), \text{ since } c < 1.
\end{aligned}$$

□

3.2 Skip Lists

We will now look at another data structure for dictionaries that has the additional property that the data is stored as an ordered lists. Even though this can also be achieved using balanced BSTs (which we will see later), skip lists are much more simpler to implement and maintain. They have poorer

See the Wikipedia page for various applications skips lists have been used for.

worst-case guarantees, but achieve average-case guarantees as good as BSTs using randomization. These were first described by William Pugh[3] in 1990.

[3] William Pugh. "Skip lists: a probabilistic alternative to balanced trees". In: *Commun. ACM* 33.6 (1990), pp. 668–676. ISSN: 0001-0782. DOI: 10.1145/78973.78977. URL: https://doi.org/10.1145/78973.78977.

"...Because these data structures are linked lists with extra pointers that skip intermediate nodes, I named them *skip lists*."

- William Pugh

Skips lists can be thought of as a sequence of lists $L_0, L_1, \dots, L_r$, where the list L_i contains a subset of the elements of L_{i-1}. The list L_0 contains the entire collection of elements and they are listed in sorted order. For a particular key k in the set, the *height* of the element is the largest i such that $k \in L_i$. If the height of an element is i, then that element is present in the lists $L_0, L_1, \dots, L_i$. Another way to think of a skip list is a list of nodes, where the node of height i has i pointers, one each to the next node in $L_i, L_{i-1}, \dots, L_0$. Each node u in the skip list contains the key associated with the node, and an array next of pointers where the size of the array is the height of u and u.next$[i]$ points to the next node in L_i. The *height of the skip list* is the maximum height among all the nodes in the skip list. The head node of the skip list has no key value, but has a pointer to the first node of each of the lists $L_0, L_1, \dots, L_h$, where h is the height of the skip list. Before we talk about how the lists $L_1, L_2, \dots, L_r$ are created, we will describe the method to implement Search, Insert and Delete.

The search starts from the head node at level equal to the height of the skip list. If the next pointer points to a value less than the key that is searched, we take that pointer to move forward. Otherwise, we moved down to the list below it. When the outer while-loop exits, then we are in L_0, and the key of the node that we are in is the largest k' in the skip list such that $k' < k$. So, we check if the next element in L_0 is the key k or not. The pseudo-code is given as Algorithm 3.

Algorithm 3.1: Search(k)

```
Node $u \leftarrow$ head
$\ell \leftarrow$ height
while $\ell \geq 0$ do
    while $u.next[\ell] \neq null$ and $u.next[\ell].key < k$ do
        $u \leftarrow u$.next$[\ell]$
    $\ell \leftarrow \ell - 1$
if $u.next[0] = null$ or $u.next[0].key \neq k$ then
    returnfalse
return $u$.next$[0].value$
```

Notice that the running time depends both on the number of pointers that we have to follow towards the key value, as well as the height of the skip list. So we want to have a situation where the height is not too large, and we can manage to skip a lot of nodes in one go at higher levels. Deterministically deciding the height of the various nodes would make insetion very expensive because if we insert in the middle we might have to change the levels of a lot other nodes as well.

To describe the procedure to insert an element into the skip list, we will assume that there is a function getHeight() that returns a number. We will not worry about how this function generates the height for now. The idea for insertion is similar to search wherein we keep moving horizontally (following the next pointer in the same list) or vertically (going to a list at the lower level) while keeping track of the nodes that we are visiting during the process. Once we reach the key largest key $k' < k$, where k is the key we are inserting, we will use the getHeight function to obtain the height ℓ of the node for key k, and insert k into the lists $L_0, L_1, \ldots, L_\ell$ while backtracking through the nodes we visited while searching.

Algorithm 3.2: Insert(k)

```
Node $u \leftarrow$ head
$\ell \leftarrow$ height
while $\ell \geq 0$ do
    while $u.next[\ell] \neq null$ and $u.next[\ell].key < k$ do
        $u \leftarrow u.$next$[\ell]$
    path$[\ell] \leftarrow u$
    $\ell \leftarrow \ell - 1$
$h \leftarrow$ getHeight()
$u' \leftarrow$ newNode$(k, h)$
while $height \leq h$ do
    height++
    path[height] $\leftarrow$ head
while $\ell \leq h$ do
    $u'$.next$[\ell] \leftarrow$ path$[\ell]$.next$[\ell]$
    path$[\ell]$.next$[\ell] \leftarrow u'$
```

Deletion of an element in the skip list is also quite similar to insertion. We would traverse the list from the head of L_h where h is the height of the skip list. We keep moving right if the key value in the next node is smaller than the key we are trying to delete. We move to a lower level if the key value of the next node is greater than the key we are trying to delete. Once we reach a list L_i and a node u such that $u.\text{next}[i].key$ is equal to k, we know that the height of the node containing k is i, and all the lists from L_i to L_0 contain k. Now, we can keep moving down while bypassing the node with key k in all these lists. The pseudo-code is quite similar to how insertion was done, except that we don't need the path pointers anymore.

Exercise 3.3. Write the pseudo-code for Delete(k) in a skip list.

The bounds on the running time will depend on the way the various lists are arranged. As we said earlier, a naive deterministic method of creating the lists can lead to potentially bad running times for insertion and deletion. We will turn to randomization for this.

3.2.1 Analysis of the running time

We will do the following random process to decide the elements of L_i that go to the list L_{i+1}. For each element $k \in L_i$, independently we will toss a coin with bias p and move k to L_{i+1} if the coin turns up head. For the ease of exposition and anlysis, we will assume for the remainder of this section that $p = 1/2$. Thus, the height of an element in the skip list is the number of heads that turn up before the first tails. This gives us the following statement about the height of any element in the skip list.

LEMMA 3.10. *Let k be any key and h_k be the height of the key in the skip list. Then, $\mathbb{E}[h_k] = 2$.*

Proof. Let E be the event that the first toss of the coin is a heads, then we can write the expectation of h_k as follows.

$$\begin{aligned}\mathbb{E}[h_k] &= (1 + \mathbb{E}[h_k])\Pr[E] + \Pr[\overline{E}] \\ &= 1 + \frac{1}{2}\mathbb{E}[h_k]\end{aligned}$$

Therefore, $\mathbb{E}[h_k] = 2$. □

We will now bound the number of nodes at any level of the skip list.

LEMMA 3.11. *For any $r \geq 0$, $\mathbb{E}[|L_r|] = \frac{n}{2^r}$ where n is the number of elements in the skip list.*

Proof. Let X_i denote the indicator random variable that is 1 when the element i in the list L_0 is present in L_r. The expectation $\mathbb{E}[X_i] = \Pr[X_i = 1] = 1/2^r$.

Now, $|L_r| = \sum_{i=1}^{n} X_i$ and hence $\mathbb{E}[|L_r|] = n/2^r$. □

While the expected heigh of individual elements is $\Theta(1)$, the height of the skip list is $\Theta(\log n)$. We show that next.

LEMMA 3.12. *If h is the height of a skip list with n elements, then $\mathbb{E}[h] \leq \log n + 2$.*

Proof. Let X_r denote the indicator random variable that is 1 when $|L_r| > 0$. We can say that $h = \sum_{r \geq 0} X_r$.

Clearly, $\Pr[|L_r| > 0] \leq 1$. Since $\mathbb{E}[|L_r|] = \sum_{i=1}^{n} i \cdot \Pr[|L_r| = i]$, we have $\Pr[|L_r| > 0] \leq \mathbb{E}[|L_r|] = n/2^r$. Thus, $\mathbb{E}[X_r] = \Pr[|L_r| > 0] \leq \min\{1, n/2^r\}$.

Therefore, we have

$$\mathbb{E}[h] \leq \sum_{r=0}^{\log n - 1} 1 + \sum_{r \geq \log n} \frac{n}{2^r} \leq \log n + 2$$

□

EXERCISE 3.4. If h is the height of the skip list, show that $\Pr[h \geq 2\log n] \leq 1/n^2$.

We will now bound the running time for searching an element in the skip list. Note that this is also the bound for inserting into a skip list and deleting an element from the skip list.

LEMMA 3.13. *The expected search time for any element in a skip list with n elements is at most $2\log n + O(1)$.*

Proof. Suppose that the height of the skip list is h. The search starts at the head node at height h. It then moves either right (if the next element at that level is smaller than the key that we are searching for) or down. The search ends in L_0 at the largest element k' such that $k' < k$. We will analyze this search path from k' going towards the head node in L_h.

If the node with key k' has height ℓ, then the last step in the search would have been a down move. In particular, if the search reaches a node at level i, and the height of the node is greater than i, then the previous step in the Search procedure would have been down move. Alternately, if a node u has height ℓ, then the Search must first enter the node at L_ℓ.

Verify this by looking at the pseudocode for Search

Since each node in level L_i moves to L_{i+1} with probability $1/2$, we can think of this reverse path as a random walk starting from k', where with probability $1/2$, the walk moves up a step, and with probability $1/2$ the walk moves left a step. Thus the expected search time is upper bounded by the expected number of steps by the random walk to reach the head node at L_h.

For a level i, let T_i be the number of steps in the reverse path that stays in L_i before it moves up. We can easily see that $\mathbb{E}[T_i] = 2$ for every i. Hence, for the expected number of steps for the reverse path to reach a node at height ℓ is 2ℓ. Since the expected height of the skip list is $\log n + 2$, the expected number of steps to reach a node at maximum height is at most $2\log n + 4$. At this point, the reverse path must move left until it reaches the head node. The expected number of nodes at the maximum neight is at most 1, and this bounds the expected length of the reverse path (and hence of the search path) to be $2\log n + O(1)$. □

3.3 Binary Search Trees

In the last two sections, we dealt with randomized data structures for dictionaries. We will now look at Binary Search Trees (BST) whose operations are deterministic. We will look at a variant of the BST that has good *amortized* complexity for the Search, Insert, and Delete. We will start with the basic definitions of a BST, and then see the BST that achieve good amortized bounds.

A BST is a binary tree where each node u in the tree has an associated key u.key, and the tree satisfies the *BST property* given below.

> For every node u with left child u_ℓ and right child u_r, the key values of all the nodes in the subtree rooted at u_ℓ is at most u.key, and the key values of all the nodes in the subtree rooted at u_r is at least u.key.

A sequence of key values based on an inorder traversal of a BST will be in increasing order. We will briefly recall how Search, Insert and Delete operations can be performed on the BST.

The Search procedure for a key k starts at the root. For each node u, we check if $u.\text{key} = k$. In case it is not, we recursively search the tree rooted at u_ℓ if $u.\text{key} > k$ or recursively search the tree rooted at u_r if $u.\text{key} < k$. The search ends if we find the key k in the BST or we reach a null node of the BST.

An Insert(k) operation proceeds like Search. If the key is found in the BST, then it is not inserted. Otherwise, the search ends in a node u, and the key k is inserted as a new left child (if $u.\mathsf{key} > k$) or a right child (if $u.\mathsf{key} < k$) of u.

For a Delete(k) operation, we first search for the key k and then depending on whether k is a leaf, has a single child, or has both children, we decide how to delete k and patch up the tree to maintain the BST property. If k is a leaf node, then it can be deleted and the parent nodes pointer altered. If k is a node with a single child u, then u can be connected directly to k's parent.

If k is a node u that has both left and right children, then we first find the node that has the smallest k' such that $k' > k$. This can be done by going to k's right child, and then following the left pointers until a node with no left child is reached. This must be the node with the key k' that we are looking for. We can make $u.\mathsf{key} = k'$ and then delete the node with key k'.

Exercise 3.5. Write down the pseudo-code for the Search, Insert, and Delete operations on a BST.

Observe that if the height of the BST is h, then all three operations take $O(h)$ time. In the worst case, a BST with n elements could have height n, and thus the time for insertion, deletion and searching could be $O(n)$. This is the case, if the key values that are inserted are in the ascending order (or descending order), creating a tree that is just a path.

One way to avoid this behaviour is to keep the binary tree *balanced*. This would make the search operations easy, but create overheads for the insertion and deletion operations. There are multiple ways in which balanced BSTs are maintained, and we will look at one specific example of a balanced BST.

3.3.1 Balanced BSTs

The idea of balance of a BST can be defined in multiple ways. The various different notions of balance are defined in such a way that for a balanced BST on n nodes, the height is $O(\log n)$. This would give good bounds on the search time. The different implementations of balanced BSTs give ways to do insertions maintaining the balance property that is defined. We will start by looking at a few different definitions of balance and how they imply the $O(\log n)$-height property.

For a node u in a BST, we will denote by $h(u)$ the height of the tree rooted at u, by $n(u)$ the number of nodes in the tree rooted at u, and by $\ell(u)$ the number of leaves in the tree rooted at u.

Definition 3.14 (AVL-balance). *A BST T is said to be* AVL-balanced *if for every node $u \in T$ with left child u_ℓ and right child u_r, $|h(u_\ell) - h(u_r)| \leq 1$.*

AVL trees satisfy this balance condition. The rotations in the insertion process of AVL trees are performed to maintain this balance condition.

The following lemma shows that the height of an AVL-balanced tree is at most logarithmic in the total number of nodes.

Lemma 3.15. *A BST with n keys that is AVL-balanced has height $O(\log n)$.*

A more precise analysis will show that the height is at most $1.44 \log n$.

Proof. Let T be an AVL-balanced BST with n nodes and height h. For a height h, let n_h denote the minimum number of nodes in any AVL-balanced BST of height h. Therefore, $n_h \leq n$.

Let T' be any AVL-balanced BST with height h and n_h nodes. For the root u of T', the height of the subtree of one of its children must be $h-1$ and the other must be $h-2$. Otherwise, if both the children had height $h-1$, we could delete one node from that tree and get a tree with less that n_h many nodes. Furthermore, the number of nodes in the two children must be n_{h-1} and n_{h-2} (minimal AVL-balanced trees). Thus, we have

$$n \geq n_h = 1 + n_{h-1} + n_{h-2}$$

Clearly, $n_{h-1} > n_{h-2}$, and hence we have $n_h \geq 2n_{h-2}$. Unrolling this recurrence, we get that $n_h \geq 2^i n_{h-2i}$. Thus, $n_h \geq 2^{h/2}$, and hence $h = O(\log n)$. □

... ignoring some floors, ceilings and corner cases.

Another notion of balance (and one that will be useful for the BST that we study later) is weight balance that is defined as follows.

... $\alpha \leq 1/2$ for this definition to make sense.

Definition 3.16 (α-weight balance). *A BST T is said to be α-weight balanced if for every node $u \in T$ with left child u_1 and right child u_2, $n(u_1) \geq \alpha \cdot n(u)$ and $n(u_2) \geq \alpha \cdot n(u)$.*

The following lemma gives a bound on the height of α-weight balanced BSTs.

Lemma 3.17. *An α-weight balanced BST with n nodes has height at most $\frac{\log n}{\log(1/(1-\alpha))}$.*

Proof. Consider the longest path in the BST T starting from the root u. Since the tree is α-weight balanced, we have $n(u_1), n(u_2) \leq (1-\alpha) \cdot n(u)$ where u_1 and u_2 are the left and right children of u. At every step of the longest path from u, the number of nodes in the subtree is at most $(1-\alpha)$ of what was present before it. Thus if we are at a node u' after i steps in this path, then $n(u') \leq (1-\alpha)^i n(u)$. Thus, at the final step, we have $1 \leq (1-\alpha)^h n(u)$. Therefore, $h \leq \frac{\log n}{\log(1/(1-\alpha))}$. □

The goal of all the different definitions of balance is to maintain a height of $O(\log n)$. One way to do this would be to maintain the balance condition after every insertion, thus getting a running time of $O(\log n)$ for Search in the worst-case. The first such construction of a balanced binary search tree was due to two Russian computer scientists, Adelson-Velski and Landis.

3.3.2 AVL trees

As mentioned above an AVL tree is a binary search tree that maintain the AVL height balance (Definition 3.14) after every operation. From Lemma 3.15, we know that this will ensure that the search operation will take $O(\log n)$ time in the worst-case. We will first see how an insertion can be performed on an AVL tree.

Insertions

Suppose that we have an AVL tree T and we insert an element x. Each node in the AVL tree can maintain an additional variable to store the height of the subtree rooted at that node. The first phase of the insertion is similar to what we do for a normal BST. We will search the BST to find the leaf node where x is going to be inserted. After insertion, we will retrace the path while updating the height variable of each node in the path from x to the root. The only nodes that may possibly have a violation of the AVL balance condition are those on the path from x t o the root. Hence, if there are no violations on any of these nodes, then the insertion stops at this point.

We will interchageably use x to denote both the key and value for now.

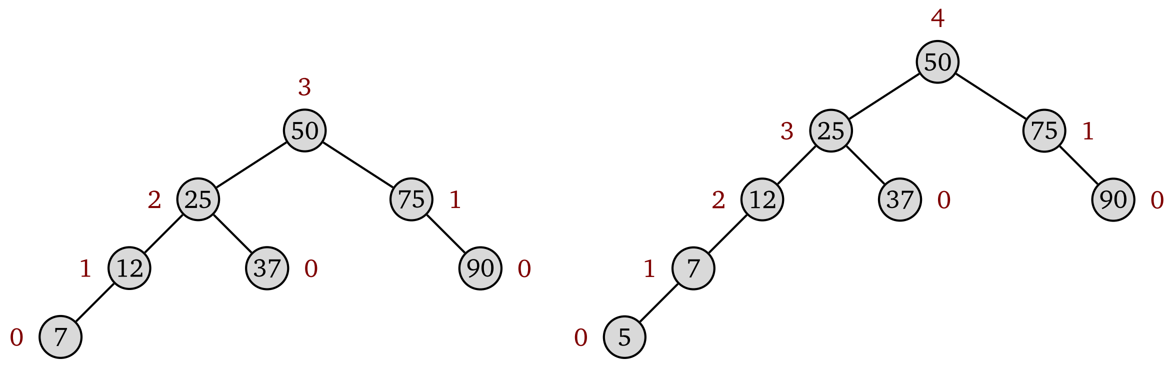


Figure 3.2: Insertion of element 5 into an AVL tree. The number inside the node is the key and the number next to the node (in red) is the height. After the insertion of 10, the nodes 12, 25, and 50 all violate the height balance condition.

The more interesting case is when there is a node u in the path from x to the root such that the absolute value of the difference between the height of its two children is more than 1. Figure 3.2 shows how the height balance condition is violated at multiple nodes on a path after an insertion.

Let u be the first node that violates the balance condition and let v and w be its two children with v being the root of the subtree where x is inserted. Furthermore, let z be the child of v in whose subtree x was inserted. What we know is that $|h(v) - h(w)| > 1$. The AVL tree rebalances itself by doing certain "rotations" of the nodes of the trees. The number and type of rotations that it performs depends on whether v and z are left/right childs of their parents. We will look at four cases, two of which are symmetric to the other two. We will call these cases as zig-zig, zig-zag, zag-zig, and zag-zag. For instance, in the example in Figure 3.2, the insertion creates an instance of zig-zig: The first node u where the height is not balanced is the node 12, and the node v where the new node is inserted is its left child, and the subtree of v where the new node is inserted is the left child of v.

We will consider both the zig-zig and the zig-zag cases. The other two are symmetric.

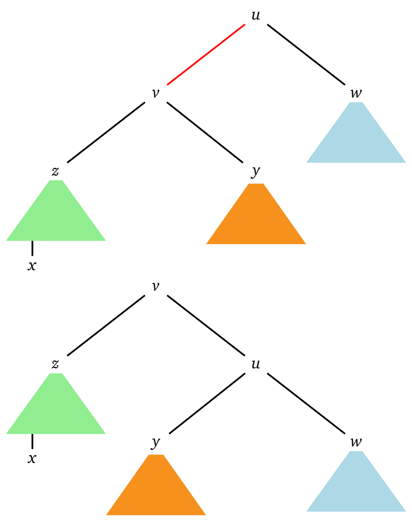


Figure 3.3: The zig-zig rotation performed when an element x is inserted into an AVL tree that causes a violation of the height balance condition. The first node where the height balance is violated is the node u. The rotation is performed about the red edge in the figure.

- **zig-zig** case: Figure 3.3 describes this case where the node that is inserted is denoted by x.

  In this case v is the left child of u and x is in the left subtree of v, rooted at z. In this case, we perform a rotation of the edge connecting v with the root u. Thus, we have v connected to the parent of u. The node u is now the right child of u and w is the right child of u. The right subtree of v in the original tree now becomes the left subtree of u, and the subtree rooted at z remains as the left subtree of v. In the specific case of the example in Figure 3.2, the rotation creates the following tree.

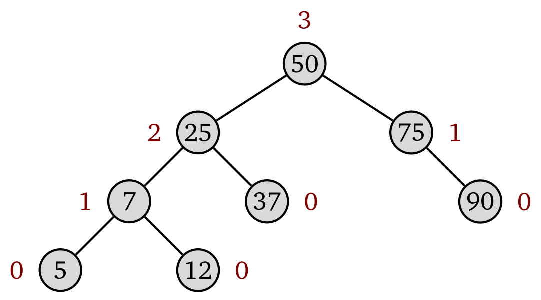


Figure 3.4: Rotation after insertion of 5 from the example in Figure 3.2. Now all the nodes are height balanced.

The rotation that is performed on the edge connecting u and v can be done by changing the pointers of the nodes associated with the tree. Observe that these pointer operations can be performed in $O(1)$ time. The corresponding operations when z is the right child of v and v is the right child of u is the zag-zag case, and you can verify that it is symmetric to the case described above. The following exercise will convince you that these operations take $O(1)$-time.

Exercise 3.6. Write down the pseudocode of rotations to be performed in the zig-zig case and the zag-zag case.

Let us now prove that this one zig-zig rotation brings back the height balance for the *entire*BST.

Claim 3.18. *If the insertion of x creates a zig-zig rotation, then the tree T is height balanced after the rotation.*

Proof. Let $h(\cdot)$ denote the height of a node before the insertion and $h'(\cdot)$ denote the height of a node after the insertion of x. We will denote the root of the right subtree of v before the rotation (refer to Figure 3.3) by y. Since the insertion of x created a zig-zig rotation, the height of v must have increased after the insertion. Therefore, $h'(v) = h(v) + 1$ since the height can increase by at most 1 after an insertion.

This insertion created a height inbalance in the node u. Therefore, it must have been the case that $h(v) = h(w) + 1$ since otherwise even if the height of v increases after the insertion, the absolute value of the difference would have remained at most 1. Also, $h(u) = h(v) + 1 = h(w) + 2$.

Let us now compare $h(z)$ and $h(y)$. If $h(z) = h(y) - 1$, then $h(v) = h(y) + 1$. Therefore, even if $h'(z) = h(z) + 1$, the height $h'(v)$ would have remained unchanged as $h(y) + 1$. Also, if $h(z) = h(y) + 1$, then $h'(z) = h(z) + 1$ would create a height inbalance at node y. But, we know that u is the first node in the path from x to the root that has a height inbalance. Therefore, it must be the case that $h(z) = h(y)$. Combined with the observation in the previous paragraph, this means that $h(z) = h(y) = h(w) = h(v) - 1$ and $h(u) = h(v) + 1$.

After the rotation $h'(y) = h'(w)$ since these heights are unchanged and hence u is height balanced, where $h'(u) = h(y) + 1 = h(w) + 1$. Also, $h'(z) = h(z) + 1 = h(w) + 1 = h'(u)$. Consequently, v is height balanced, and $h'(v) = h'(u) + 1 = h(w) + 2$.

Thus, all the subtrees below v are height balanced. Furthermore, new height of v is equal to the height of the node u before x was inserted. Since that tree was height balanced for all nodes, all the nodes above v in the rotated tree are also height balanced. □

The analysis shows that in the zig-zig/zag-zag case, the tree can be rebalanced with just one rotation. Hence the time of insertion in this case is $O(\log n)$.

- **zig-zag** case: Figure 3.5 shows the zig-zag case where x is the new node that is inserted.

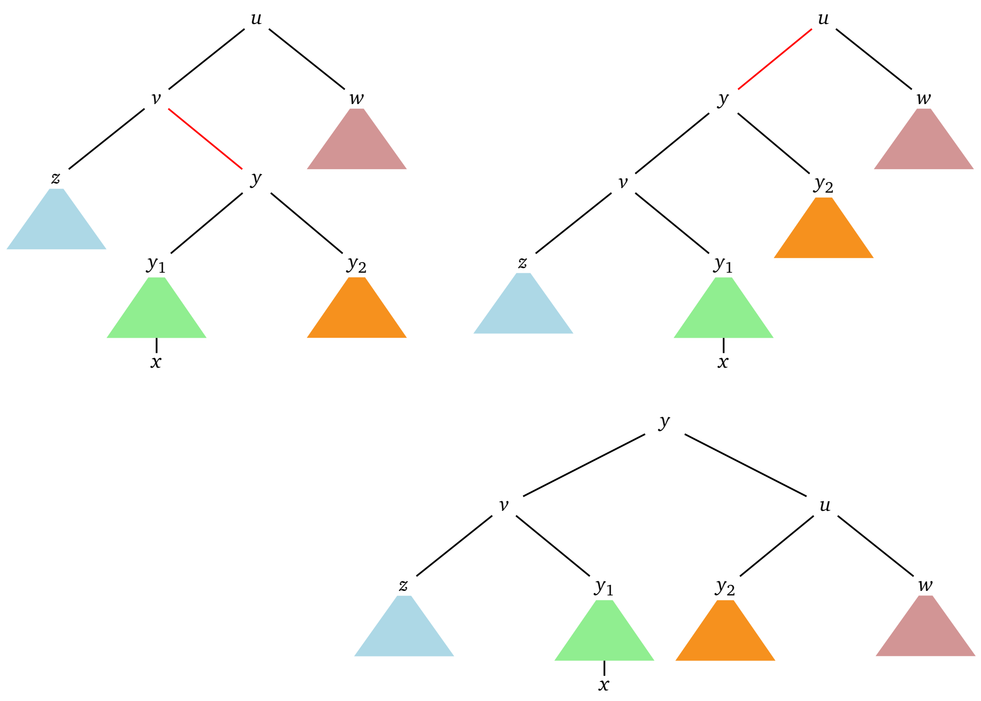


FIGURE 3.5: The insertion of the element x creates a height inbalance and u is the first node in the path to the root that violates the height balance. This is corrected by performing two rotations as shown here. The red edge is the edge about which the rotation is performed.

There are two rotations that we perform in the zig-zag/zag-zig case. Since each operation takes $O(1)$ time, and the search takes $O(\log n)$ time, we can say that the insertion takes $O(\log n)$ time, provided we can show that the tree is height-balanced at each node. The proof is analogous to the previous case, but we just state it for completeness.

CLAIM 3.19. *If the insertion of x creates a zig-zag case, then the height is rebalanced after performing the two rotations described above.*

Try proving this fact before reading the proof.

Proof. As before, assume that $h(\cdot)$ is the height before the insertion of x and $h'(\cdot)$ is the height after the insertion of x and the two rotations. Since u is the first node that violates the balance condition, it must be the case that $h(v) = h(w) + 1$ and that the insertion of x increased the height of v. Similarly, it must be the case that $h(y_1) = h(y_2)$ since if $h(y_1) = h(y_2) - 1$, then insertion x will not cause any increase in the height of y and hence the height of v will remain unchanged. On the other hand, if $h(y_1) = h(y_2) + 1$, then if the insertion of x does not increase the height of y_1, the height of v will remain unchanged. But, if the insertion of x increases the height of y_1, then y will be the first node in the path from x to the root that is unbalanced and this contradicts the assumption that u was the first such vertex. The same reasoning also shows that $h(z) = h(y) = h(y_1) + 1 = h(y_2) + 1$. Therefore, $h(v) = h(z) + 1 = h(y) + 1$, and hence $h(z) = h(y) = h(w)$.

Now, let us look at the tree after the completion of the two rotations. From the observations above, we can see that $h(y_2) = h(w) - 1$ and hence $h'(u) = h(w) + 1$. Therefore, u is now height balanced. Similarly, $h(z) = h(y_1) + 1 = h'(y_1)$, and therefore v is height balanced and $h'(v) = h(z) + 1$. Since $h(z) = h(w)$, we can conclude that $h'(v) = h'(u)$ and hence y is also height balanced. Finally, $h'(y) = h'(v) + 1 = h(w) + 2 = h(u)$. Therfore, the subtree rooted at y after the insertion and rotations has the same height as the tree rooted at u before the insertion of x. Hence the final tree is also height balanced at all the other nodes. □

The case when the inserted node is in subtree rooted at y_2 is symmetrical. The zag-zig case is also symmetric, with the only difference being the direction of the rotations.

Exercise 3.7. Describe the insertion algorithm for the zag-zag and zag-zig cases, and verify that the rotations create a height balanced tree.

We will now look at implementing deletions. They are also similar to insertion, but it is possible that multiple rotations along a leaf to root path may have to be performed to maintain the balance. This would still be fine since any path is of length $O(\log n)$ and each rotation can be performed in $O(1)$ time.

Deletions

Suppose that a node x is deleted from an AVL tree. We will first perform the same deletion algorithm as in the case of a normal binary search tree. Let u be the first node in the path containing x that violates the height balance property. Observe that u could be a descendent of x since we replace x with its inorder successor when x has both its children present. So it is possible that one of the ancestors of the inorder successor is the node u - see Figure 3.6. In any case, we can find the node u in $O(\log n)$ time. We will describe the rotations performed by the algorithm using an example, before stating the general case.

Let v and w denote the left and right children of u, respectively. Since $|h(v) - h(w)| > 1$, we will consider the child that has greater height and try to balance on that side. Since rotations can only reduce heights, we are better off trying to reduce the height of the side that is taller! Once again

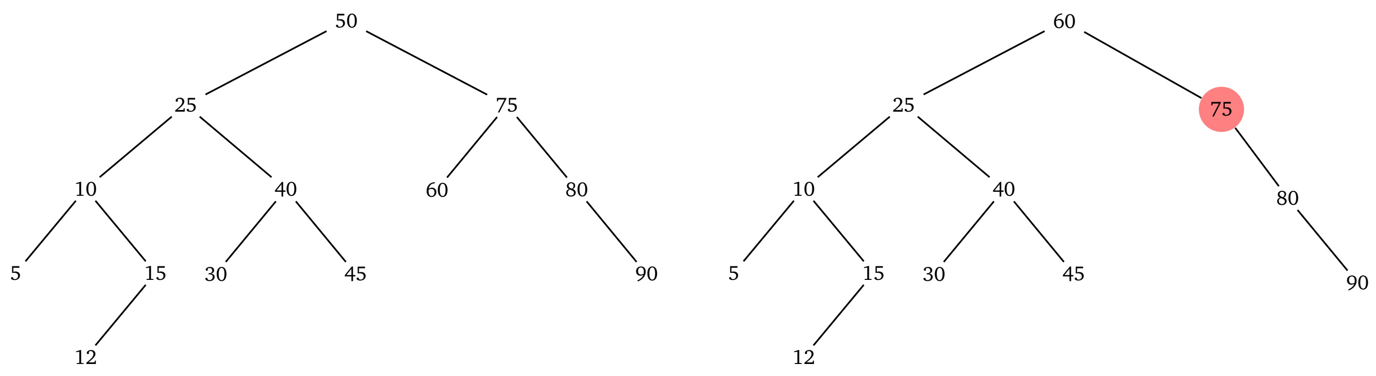


Figure 3.6: AVL tree where the deletion of the key 50 creates an unbalanced node that is the parent of the inorder successor of 50. The first node with the height inbalance is shown in red.

there are multiple cases to consider. For now assume that w has the greater height. Let y and z denote the left and right children of w. In the example in Figure 3.6, the node u is 75, the node w is 80 and v is an empty node. Among the children, y is empty and z is the node 90. Among the children of w, the node with greater height is chosen to decide if we will do a zag-zag or zag-zig rotation. If the the two heights are the same, we will choose the the right child, since w is a right child. This way we will need to perform just a single rotation.

In the example, the first rotation that is performed is a zag-zag rotation about the edge connecting 75 and 80 that gives the first tree in Figure 3.7. Now, the node 60 is not height balanced, and the child 25 has the greater height. Also, the child of 25 with greater height among the children is the left child 10. This leads to a zig-zig rotation about the edge connecting 60 and 25 to get the final AVL tree.

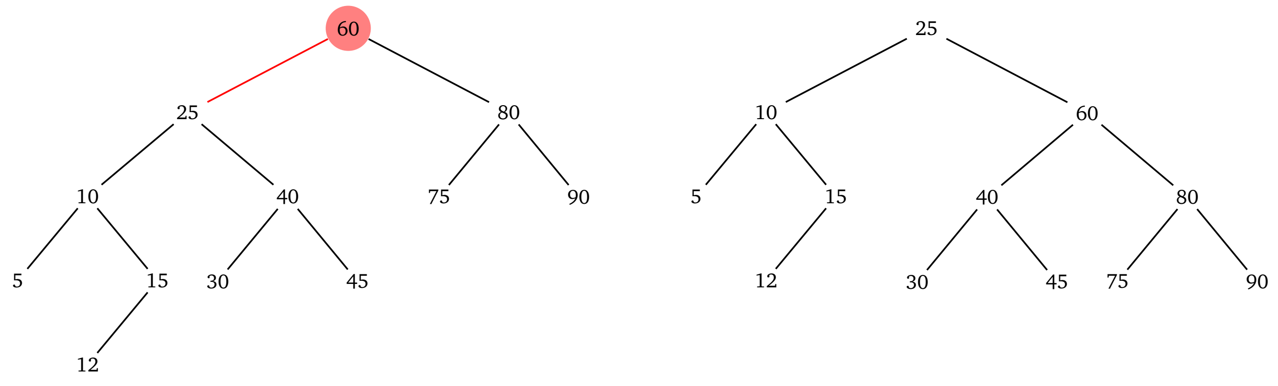


Figure 3.7: The zag-zag rotation about the edge between 75 and 80 creates a new unbalanced node that is shown in red. The edge over which the next rotation is to be performed is also shown in red.

In the general case, let u be a node that violates the height balance condition. Let v and w be its left and right child, respectively. Assume that $|h(v) - h(w)| > 1$; the other case is symmetric. Since the height inbalance came about due to a deletion of one node, it must be the case that $h(v) - h(w) = 2$. If y and z are the left and right children of v, respectively, then if $h(y) \geq h(z)$ then this becomes a zig-zig case. When $h(y) < h(z)$, we get the zig-zag case. Rotations do not affect the height balance of any nodes apart from the ancestors of the node u. Therefore, at most $O(\log n)$ rotations are sufficient to rebalance the entire tree after a deletion.

Exercise 3.8. Describe how deletion is performed on an AVL tree, and prove that zig-zig, zig-zag, zag-zag, and zag-zig rotations rebalances the tree at node u.

This construction that we saw just now, requires maintaining auxiliary information about the height and performing the rotations even if the height balance at any node is affected. We are going to look at a construction, that does not try to maintain the balance at all times, but makes sure that the height is at most $\log n$. Thus, we will get a construction where the *amortized* complexity of Insert and Delete are small, even though there could be $O(n)$-time operations in the worst-case.

3.3.3 Scapegoat trees

Scapegoat[4,5] trees are BSTs that have the property that the height of the BST is $O(\log n)$ always. The BST will not necessarily be balanced all the time, but we do lazy rebalancings to maintain the property that the height is at most $O(\log n)$. The rebalancing is done by completely or partially rebuilding a subtree to create a perfectly balanced BST. The root of the subtree that is completely rebuilt is referred to as the *scapegoat*. We will start with the following easy exercise that we will use throughtout this section.

[4] Arne Andersson. "General balanced trees". In: *Journal of Algorithms* 30.1 (1999), pp. 1–18.

[5] Igal Galperin and Ronald L Rivest. "Scapegoat trees". In: *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. 1993, pp. 165–174.

We will say that a BST is *perfectly balanced* if for every node u having left child u_1 and right child u_2, we have $|n(u_1) - n(u_2)| \leq 1$.

Exercise 3.9. Given an arbitrary BST with n keys, give an $O(n)$-time algorithm to construct a perfectly balanced BST on those n keys.

3.3.3.1 Deletions - global rebuilds

Suppose that you have a BST with n nodes that is $\frac{1}{3}$-weight balanced, and we have only search and delete operations to be performed on this BST. It is possible to create a worst-case sequence of deletions so that the after deleting some αn nodes, the time for searching shoots up to $O(n)$. Is there anything better that you can do if you knew beforehand that only search and delete operations are to be performed?

Suppose that whenever we have a delete operation on a key k, we mark the corresponding node u with a tombstone $\top$ to indicate that the key has been deleted. If we do this, we can still perform the search operation using the same algorithm as in the BST, but the running time of search will be proportional to the number of elements in the BST plus the number of tombstones.

Let t be the number of tombstones and n the number of actual elements in the BST. We will always make sure that $n \geq t$. If after performing a deletion, we have $n < t$ (there are more tombstones than actual elements), we will perform a total rebuild of the BST to create a perfectly balanced BST (using the algorithm in the exercise) that takes $\Theta(n)$-time. Thus, the worst-case running time for deletion is $O(n)$, but we will show that the amortized running-time is only $O(\log n)$.

Suppose that we look at the BST right after a complete rebuild. Say the BST has n' elements. Now, the next total rebuild happens after $n'/2$ deletions. Each of these deletions takes $O(\log n)$ time since it only requires searching for the element and marking it with a tombstone. Thus, the total time to perform n deletions can be obtained as follows:

- $O(\log n)$ time for each search operation - which gives $O(n \log n)$ time over all n deletions.
- Time for a total rebuild is proportional to the size of the BST, and since the rebuilds happen when the number of nodes in the BST is halved, we have the total time for rebuilds as $\frac{n}{2} + \frac{n}{4} + \ldots + 1 = O(n)$.

Thus the total time to perform n deletions is $O(n \log n)$ and hence the amortized cost per deletion is $O(\log n)$.

3.3.3.2 Insertions - partial rebuilds

Now, let us look at the case when we have insertions alone. The analysis and the bounds will not change a lot if we have deletions as well, since the number of tombstones $t \leq n$ always. We saw earlier that if a BST is $1/3$-weight balanced, then its height is at most $\frac{\log n}{\log(3/2)} = \log_{3/2} n$. In the case of scapegoat trees, we will only insist that the height h of the tree always satisfies the condition that $h \leq \log_{3/2} n$; we will not insist on any weight balancing. We will perform partial rebuilds of the BST if this height condition is violated during an insertion. The partial rebuild is done on a subtree rooted at a *scapegoat node* that is defined below.

"...The goat shall bear on itself all their iniquities to a barren region; and the goat shall be set free in the wilderness"
-Leviticus 16:21-22

DEFINITION 3.20 (α-scapegoat). *Let T be a BST with n elements. A node u is said to be an α-*scapegoat *if it has a child v such that $n(v) > \alpha \cdot n(u)$.*

We will be interested in $2/3$-scapegoats in our analysis. We will use the following lemma about the existence of scapegoat nodes.

LEMMA 3.21. *Let T be a BST on n vertices and let u be a leaf at depth $h > \log_{3/2} n$. Then there exists a $\frac{2}{3}$-scapegoat on the path from u to the root r of T.*

Proof. Suppose not. Consider the path from the root $r = u_0, u_1, u_2, \ldots, u_h = u$. Then, for every i, $n(u_i) \leq \frac{2}{3} n(u_{i-1})$. Hence we have $n(u) \leq \left(\frac{2}{3}\right)^h n(u_0)$. Since $h > \log_{3/2} n$, $\left(\frac{2}{3}\right)^h < 1/n$. But then $1 = n(u) < \frac{1}{n} \cdot n = 1$, which is contradictory. □

The Insert procedure on a scapegoat tree initially follows the same steps as insertion into a BST. During the insertion, the height is calculated. If it has increased to a value greater than $\log_{3/2} n$, then we traverse the search path from the inserted node towards the root while calculating the size of the subtree rooted at each of the nodes in the path. By Lemma 3.21, we know that there is a $2/3$-scapegoat node u on this path. Once we find u, we rebuild the tree rooted at u to a perfectly balanced balanced BST - this takes time $O(n(u))$.

... which could be $O(n)$ if the scapegoat is the root of the tree.

EXERCISE 3.10. Write down the pseudo-code for insertion into a scapegoat tree. You will need a subroutine call to the algorithm that constructs a perfectly balanced BST from an arbitrary BST.

We will show that the amortized time complexity of insertions is only $O(\log n)$ even though there could be $O(n)$-time insertions in the worst-case.

LEMMA 3.22. *The total cost of n insertions into a scapegoat tree is $O(n \log n)$.*

Every search on an n-node scapegoat tree is $O(\log n)$ in the worst-case.

Proof. We will use the accounting method to calculate the amortized complexity of the operations. For every insertion of a key k into the scapegoat tree, we will give three units of credit to each of the nodes in the search path for the key k that is done before the insertion. Thus the total credits used per insertion is the actual cost of the insertion plus the $\log_{3/2} n$ credits given across all the nodes in the path.

Consider the an insertion of a key k that results in the height property getting violated. Also, let u be the $2/3$-scapegoat on the path from the node where k is inserted to the root r. Let v be the child of u such that $n(v) > \frac{2}{3}n(u)$, and let v' be the sibling of v in the tree.

We also know that $n(u) = n(v) + n(v') + 1$. If $n(v) > \frac{2}{3}n(u)$, then this means that $n(v') < \frac{1}{3}n(u) - 1$. Consequently, we have

$$n(v) - n(v') > \frac{1}{3}n(u) + 1.$$

Observe that right after the last rebalancing was done on the subtree containing u, it must have been the case that $|n(v) - n(v')| \leq 1$. Thus for $n(v) - n(v')$ to be more that $\frac{1}{3}n(u) + 1$, there must have been $> \frac{1}{3}n(u)$ insertions in the subtree rooted at u. Consequently, u has credits worth at least $n(u)$ with itself obtained during those insertions. Since the time for the rebuild is $O(n(u))$, these credits can be used for the rebuild operation. Thus, the amortized cost per insertion is $O(\log n)$. □

3.3.3.3 Putting insertions and deletions together

The modifications in the algorithms and their analysis in the case of insertions and deletions is minimal. We will not have to take care of the fact that the scapegoat tree contains nodes with keys as well as tombstones. The α-weight balances and α-scapegoats will be defined based on the total size of the tree includes nodes containing keys and scapegoats.

Let m be the number of nodes with key values, and let t be the number of tombstones in the scapegoat tree. Thus the size of the tree $n = m + t$. We will maintain the invariant that $m \geq t$. The only way that m reduces is when there is a deletion operation on the tree. At this point, we do a global rebuild. The height property maintained by the scapegoat tree will now be that $h \leq \log_{3/2} n$ (includes both actual keys and tombstones).

3.3.4 Randomized BSTs and treaps

We will look at a randomized version of BSTs that provides the $O(\log n)$ guarantees associated with BSTs, but in expectation. As a first step, let us consider the case of a set $\{a_1, a_2, \ldots, a_n\}$ of elements that we want store in a BST. The shape of the BST is determined by the order in which these elements are inserted into the tree. For instance, if $a_1 < a_2 < \cdots < a_n$ and the sequence of insertions is $a_1, a_2, \ldots, a_n$, then the BST consists of a list where each node is the righ child of its parent. On the other hand, there are sequences that lead to a more balanced search tree. The importance of balance is that the search tree operations can be performed efficiently, compared to the $O(n)$-time if the tree is skewed.

Suppose now that we permute the sequence $a_1, a_2, \ldots, a_n$ uniformly at random, and the elements are inserted into the BST in this order. What can we say about the height of the tree? Recall that the time complexity of operations on the search tree is $O(h)$ when h is the height of the tree. Instead of reasoning about the height of the tree, we will prove the weaker statement that for every key the expected search time is small.

It is also true that $\mathbb{E}[h] = O(\log n)$. Why is this a weaker statement?

Consider a key value x. Let X_j denote the indicator random variable such that

$$X_j = \begin{cases} 1 & a_j \text{ is in the search path of } x \\ 0 & \text{otherwise} \end{cases}$$

The random variable X defined as

$$X = \sum_{j \mid a_j \neq x} X_j$$

denotes the runtime for searching the key x. We are interested in $\mathbb{E}[X]$, and by the linearity of expectation it is sufficient to compute $\mathbb{E}[X_j]$. We will consider two cases based on the value of x.

- $\underline{x = a_i \text{ for some } i}$: If $j < i$, then a_j is in the search path of x iff the first element from the set $\{a_j, a_{j+1}, \ldots, a_i\}$ in the sequence is a_j. This is because if for any $k > j$, a_k appears before a_j, then a_j lies in the left subtree of a_k and a_i lies in the right subtree of a_k, and hence a_j will never be reached while searching for x. Similarly, if $j > i$, then a_j lies in the search path of x iff the first element from the set $\{a_i, a_{i+1}, \ldots, a_j\}$ in the sequence is a_j. The case when $i = j$ is trivial since $\mathbb{E}[X_j] = 1$. Thus for this case we can write the expectation as

Recall the analysis of quicksort using linearity of expectations in Section 2.3.2.

$$\mathbb{E}[X_j] = \begin{cases} \frac{1}{i-j+1} & \text{if } j \leq i \\ \frac{1}{j-i+1} & \text{if } j \geq i+1 \end{cases}$$

- $\underline{a_i < x < a_{i+1} \text{ for some } i}$: A similar argument like in the case before gives the expectation as

$$\mathbb{E}[X_j] = \begin{cases} \frac{1}{j-i+1} & \text{if } j \le i \\ \frac{1}{j-i} & \text{if } j \ge i+1 \end{cases}$$

From the expressions above, we can upper bound the value of $\mathbb{E}[X]$ for searching a key x (where $a_i \le x < a_{i+1}$) as

$$\begin{aligned}\mathbb{E}[X] &\le \sum_{j \le i} \frac{1}{i-j+1} + \sum_{j>i} \frac{1}{j-i} \\ &= \sum_{k=1}^{i} \frac{1}{k} + \sum_{k=1}^{n-i} \frac{1}{k} \le H_i + H_{n-i} \le 2H_n.\end{aligned}$$

Here H_n is the n^{th} Harmonic number where $H_n = \ln n + \Theta(1)$. Summarizing the discussion, we can state the following theorem.

Theorem 3.23. *Let $K = \{a_1, a_2, \ldots, a_n\}$ be a set of key values and let π be a permutation of $\{1, 2, \ldots, n\}$ chosen uniformly at random. Let T be the binary search tree obtained by inserting the set K in the order $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$. Then for any value x, the expected time to search for x in the tree T is at most $2\ln n + \Theta(1)$.*

This is not useful on its own from the point of view of dynamic insertions and deletions since the analysis assumes the sequence of keys beforehand. We will now see a randomized data structures that achieves these bounds.

3.3.4.1 Trees + Heaps = Treaps

The problem with using the algorithm from the previous discussion was that it required the knowledge of the set K of keys beforehand, whereas in typical scenarios what we have is a sequence of insertions, searches and deletions. Thus, we will not be able to make sure that the insertions are done in a uniformly random order of the key values. To get around this, a clever data structure leverages the ideas of BSTs and heaps. The *treap*[6] is a tree whose nodes contain a key value and a priority. The tree is BST w.r.t the key values, and satisfies the heap property w.r.t the priority values. We will show that the analysis, and the guarantees of this data structure matches what we obtained earlier if the priority values are distinct values chosen at random.

[6] Raimund Seidel and Cecilia R Aragon. "Randomized search trees". In: *Algorithmica* 16.4 (1996), pp. 464–497.

Since the treap is a BST w.r.t to the key values, we can perform the search operation just like in the case of a BST. Let us now look at the insert procedure in a treap. Consider a key value x, that has been assigned a priority p uniformly at random. We can assume that the total number of keys inserted across the entire sequence of operations is n. Thus, if we choose a priority value as a random $4\log n$-bit number, then with probablity at least $1 - 1/n^2$ all the priority values will be distinct.

Now, we will first insert the pair (x, p) in the BST using with the key value x. This insertion will be done using the BST insertion procedure. Thus, the pair (x, p) will be a leaf node in the treap. We may not be done at this stage since the parent (y, p') of (x, p) might have a priority $p' > p$ and this will destroy the heap property. To restore the heap property of the treap, we will perform rotations on the treap. There are two possible rotations depending on whether the child node is a left or right child of its parent. Figure 3.8 shows the rotations.

Notice that one right rotation amounts to changing the pointers of $O(1)$ nodes - the right child of p becomes the left child of p', p' becomes the right child of p, the parent pointer of p will now point to the parent of p', and the corresponding child pointer of p's parent will point to p. Each time a rotation is performed, notice that the depth of (x, p) reduces by 1. Therefore, the total time to complete the insertion is at most the search depth of the key x.

How does the random priorities help in bound the search time for a key x? Consider the treap formed by inserting the pairs $(a_1, p_1), (a_2, p_2), \dots, (a_n, p_n)$. Since the treap satisfies the heap property w.r.t to the p_is, the root of the treap consists of the pair (a_i, p_i) for the least priority p_i. Now, all the key values $a_j < a_i$ is in the left subtree and all the key values $a_j < a_i$ are in the right subtree.

Consider the sequence of keys $a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(n)}$ such that $p_{\pi(1)} < p_{\pi(2)} < \dots < p_{\pi(n)}$. If the keys were inserted in a BST in this order, then the shape of the BST would be precisely that of the treap that was constructed as described above. This is because the treap would contain $a_{\pi(1)}$ as the root, and this would be same when $a_{\pi(1)}$ was first in the sequence of keys being inserted. Now all the keys greater(/smaller) than $a_{\pi(1)}$ will be in right(/left) subtree of $a_{\pi(1)}$. Inductively, the root of the left subtree will be the key with the smallest priority among them. Thus, the random priorities play the role of the random ordering that we analyzed in the previous discussion.

Let us look at the case of deletion of a key value x with priority p. If x is already a leaf, then it can be deleted maintaining both the BST property on the keys and the heap property on the priorities. If x is an internal node with children y and z with priorities p' and p'', choose the $\hat{p} = \min\{p', p''\}$, and rotate the treap around that node. Keep doing this process until x becomes a leaf. Notice that (x, p) will be the only node breaking the heap property, and it can be deleted once it becomes the leaf. The bound on the running time follows from observing that the sequence of rotations performed during the delete operation is the opposite sequence of the operations when (x, p) is inserted into the treap.

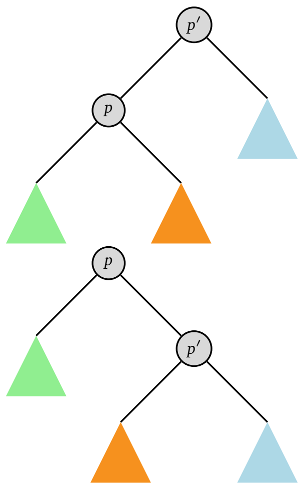


FIGURE 3.8: The first tree has priority value p' at the root and $p < p'$ destroying the heap property. Since p is a left child of p', a right rotation is performed making p' the right child of p, and the right subtree of p becoming the left subtree of p'. This maintains the BST property w.r.t the key values.

EXERCISE 3.11. Write the pseudocode for the Insert and Delete operations.

The constructions above can be summarized by the theorem below.

THEOREM 3.24. *For a treap T with n elements where the priorities are chosen uniformly at random, the Search, Insert, and Delete operations can be performed in expected $O(\log n)$-time.*

3.3.5 Splay trees

A splay tree is a self-balancing search tree, much like the scapegoat tree, that we saw earlier. It was described first by Sleator and Tarjan,[7] and supports all

[7] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-adjusting binary search trees". In: *Journal of the ACM (JACM)* 32.3 (1985), pp. 652–686.

the BST operations in $O(\log n)$ amortized time. Unlike the scapegoat tree, the maximum height of a splay tree can be as bad as $O(n)$. The basic operation in a splay tree is the *splay* operation, that moves the most recently accessed element to the root of the tree via a series of rotations. With splaying, the most recently accessed elements are near the root of the BST, and hence can be accessed quickly.

The splaying operation is performed by a sequence of rotations. Any element x in the tree can be moved to the root by a sequence of rotations. Instead of doing these sequences of rotations only on the node x, the splay tree performs a sequence of two rotations on x and its parent (unless the parent is already the root). We will classify the rotations as *zig*, *zag*, *zig-zag*, *zag-zig*, *zig-zig*, *zag-zag*. The operations zig and zag are symmetric, so are zig-zag and zag-zig, and zig-zig and zag-zag.

We will see each of these operations with examples.

1. The **zig** operation: This operation is perfomed on a node x, that is the left child of its parent $p(x)$ where $p(x)$ is the root of the tree.

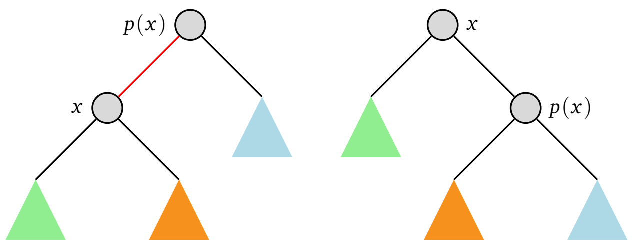


FIGURE 3.9: The zig rotation performed about the red edge.

2. The **zag** operation is symmetric when x is the right child of its parent $p(x)$ and $p(x)$ is the root of the tree.

3. The **zig-zig** operation is performed when accessing a node x which is the left child of its parent $p(x)$, and $p(x)$ is also the left child of the grandparent of x (denoted by $g(x)$). In this case, the edge between $p(x)$ and $g(x)$ is rotated, followed by the edge between x and $p(x)$.

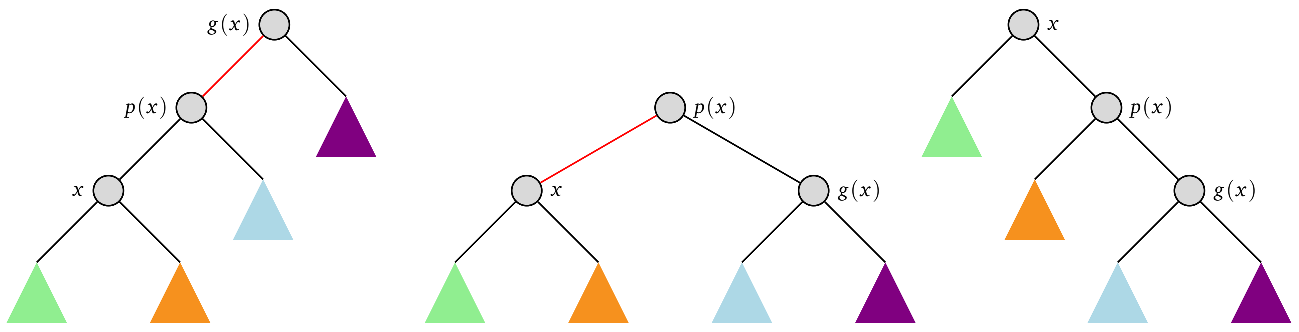


FIGURE 3.10: The zig-zig rotations consists of two rotations - first performed about the edge connecting $p(x)$ with $g(x)$ and then about the edge connecting x and $p(x)$. The edges are drawn in red.

4. The **zag-zag** operation is symmetric to the zig-zig operation when x and $p(x)$ are the right children of their respective parents. After a zig-zig/zag-zag operation the depth of x reduces by 2, the depth of $p(x)$ remains unchanged, and the depth of $g(x)$ increases by 2.

5. The **zig-zag** operation is performed when accessing a node x that is the right child of its parent $p(x)$, and $p(x)$ is the left child of x's grandparent $g(x)$. First, the edge between x and $p(x)$ is rotated, followed by the edge between x and its new parent, which will be $g(x)$.

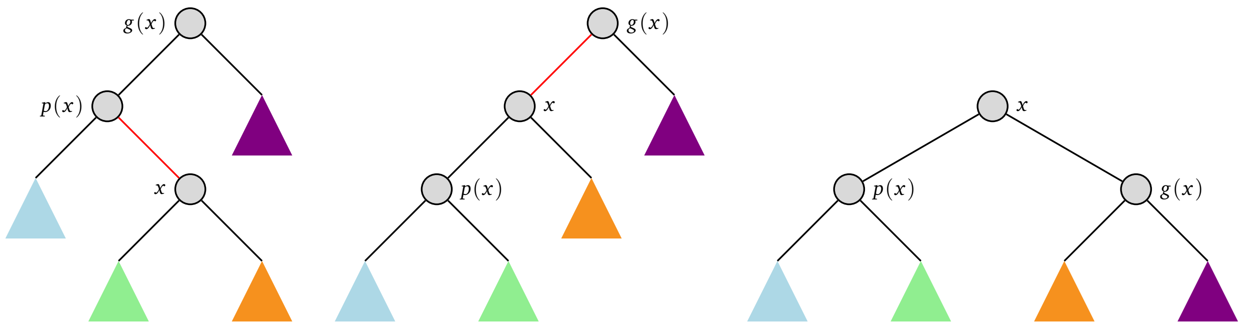


FIGURE 3.11: The zig-zag operation is performed first about the edge connecting x with $p(x)$, and then about the edge connecting x with $g(x)$. The edges are drawn in red.

6. The **zag-zig** operation is symmetric to the zig-zag operation that is described above.

Each of these operations takes $O(1)$ time as they consist of at most two rotations. The total time for splaying is $O(d_x)$ where d_x is the depth of the node x in the search tree.

4 Priority Queues

A priority queue is an ADT that stores a set of key values each of which has an associated priority value. For the ease of presentations, we will assume that the key values themselves are priorities. A priority queue Q supports the following operations.

- $\mathsf{Insert}(k)$ - insert the key k into Q.
- GetMin - returns the smallest key k.
- $\mathsf{ExtractMin}$ - returns the smallest key k and delete it from Q.
- $\mathsf{DecreaseKey}(k, k')$ - reduce the key value k to k' (assuming $k' < k$).

Analogous operations can be defined if we are interested in maintaining the obtaining the maximum element or increasing key values. After looking at simple implementations of priority queues, we will also look at modifications that allow the Meld operation that combines two priority queues to create a new priority queue efficiently.

4.1 Binary minheap

A simple data structure that implements the priority queue operations defined above is the binary minheap. We will briefly recall the operations and time-complexity of the binary minheap before we look at more complicated data structures.

A binary minheap is a binary tree that satisfies the following two conditions.

- The binary tree corresponding to the minheap is a full binary tree - i.e. all the levels except the last level are complete.
- For each node u in the minheap, the key $u.\mathsf{key}$ is less than the keys values of its children.

Binary minheaps are represented using an array Arr, indexed from 1 to n. The array can be thought of as the BFS traversal of the heap where the children of each node are listed from left to right. Thus $\mathsf{Arr}[1]$ is the root of the minheap, and for any $i \leq n/2$, $\mathsf{Arr}[2i]$ and $\mathsf{Arr}[2i+1]$ are its left and right children. From now on, we will identify the heap and array representing it by the same symbol.

Exercise 4.1. Verify that a binary minheap with n elements has height $\log n$.

Let H be a minheap of size n, and let k be a new key that is being inserted into the heap. We start by adding k into the position $H[n+1]$, and *bubbling up* the element k to mainatain the heap property. Observe that k is inserted as a child of the element in $H[(n+1)/2]$. If $k < H[(n+1)/2]$, then k is swapped with that element. We then keep continuing until k cannot be bubbled up any further. Thus insertion of a key into a minheap has time complexity equal to the height of the heap, which is $O(\log n)$.

We will assume that the array in which the heap has sufficient space for insertions, and will not worry about the problem of resizing.

The other key operation on a minheap is the *trickle down* operation which is performed when the minimum key is extracted from the heap. Recall that the minimum element in a minheap is present in $H[1]$. To remove this element, the element $H[n]$ is copied to $H[1]$ and the size of heap is reduced to $n-1$. The element $H[1]$ is swapped with the smallest element in the set $\{H[1], H[2], H[3]\}$. If $H[1]$ is the smallest element in the set, then the extraction operation is finished. Else, the element is trickled down by performing the swaps repeatedly. Once again, the time complexity of extraction is at most the height of the heap, and hence $O(\log n)$.

The DecreaseKey operation is performed by first reducing the key value to k' and then bubbling it up until the heap property is satisfied. This operation has a time complexity of $O(\log n)$ in the worst-case.

Exercise 4.2. Write down the pseudo-code for the priority queue operations when the queue is implemented as a binary minheap.

We will end this revision by recalling the algorithm to create a minheap from an arbitrary array of n elements. One straightforward method is to keep inserting the elements in the array into the heap using the algorithm we have seen earlier. Since each insert operation has a time-complexity of $O(\log n)$, the total time to create the minheap is $O(n \log n)$. A more efficient algorithm is to think of the array as a binary tree that is complete, except for the final level, and then to make the tree satisfy the heap property starting from the nodes at the maximum depth.

For an array H, starting at $H[n/2]$ we keep making the subtree rooted at that node into a minheap. This would require trickling down the root until the minheap property is satisfied. Consequently, the number of steps to *heapify* the subtree rooted at a node u at height h is $O(h)$. If $N(h)$ is the number of nodes at height h, then the total running time of creating the heap is given by the sum

$$\sum_{h=0}^{\log n} N(h) \cdot O(h).$$

To complete the analysis, we use the following lemma which is left as an exercise.

Lemma 4.1. *The number of nodes in a minheap (of total size n) at height h is at most $n/2^h$.*

Thus, the cost of converting an array of size n into a minheap is at most

$$\sum_{h=0}^{\log n} N(h) \cdot O(h) \leq \sum_{h=0}^{\log n} \frac{n}{2^h} c \cdot h \leq c \cdot n \sum_{h=0}^{\log n} \frac{h}{2^h} = O(n).$$

4.2 Min-max heaps

A small modification of the minheap structure gives you the benefit of both maxheap and minheap. This is the *min-max heap*[1] and it supports Insert, ExtractMin, and ExtractMax in $O(\log n)$-time.

[1] Michael D Atkinson et al. "Min-max heaps and generalized priority queues". In: *Communications of the ACM* 29.10 (1986), pp. 996–1000.

The levels of a min-max heap alternate between max-levels and min-levels, starting with the level zero, which is a min-level. For a node u in a min-level, the key value is the smallest among all its descendants. Similarly, for a node u in the max-level, the key value is the largest among all its descendants.

In a min-max heap, the smallest element is at the root of the heap, and the largest element is one of its two children. In the figure given to the right the minimum element is 3 (at the root), and the maximum element is 40 (the left child of the root). Before we describe the insertion and extraction algorithms, we will explain how the bubbling up and trickling down operations are performed. They are quite similar to the operations for an ordinary minheap; we will have to keep the levels also in mind while performing this operation on a min-max heap.

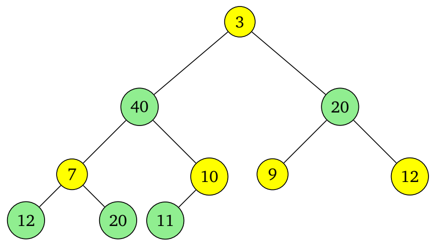


FIGURE 4.1: A min-max heap of height 3. The yellow colored levels are the min-levels and the green colored levels are the max-levels.

The pseudo-code for this trickle down operation is given below. This is the algorithm for trickling an element down when it is in a min-level.

Algorithm 4.1: TrickleDownMin(i)

```
//indices of children and grandchildren
$S \leftarrow \{2i, 2i+1, 4i, 4i+1, 4i+2, 4i+3\}$
if $\forall i \in S$, $H[j] = null$ then return
$m \leftarrow \arg\min_{j \in S} H[j]$ //index of the smallest element in $S$
if $H[i]$ is the smallest then return
Swap key values of $H[i]$ and $H[m]$
if $m$ was a grandchild of $i$ then
    if $H[m] > H[m/2]$ then
        Swap key values of $H[m]$ and $H[m/2]$
TrickleDownMin($m$)
```

Consider the case where we have an element at the root (which is a min-level) where the heap property is not satisfied. Assume that the subtree rooted at the children of the root are max-min heaps (i.e. the root level is a max-level). We first check if the key at the root is indeed the smallest. Since the subtrees of the root all satisfy the heap property, and since we are at a min-level, we need to check for the smallest value among the grandchildren (if they exist) of the node; if there are not grandchildren, we have to check with the children. If the smallest value is at position i, then the key at the root is swapped with the element at position i. After this swap it is possible

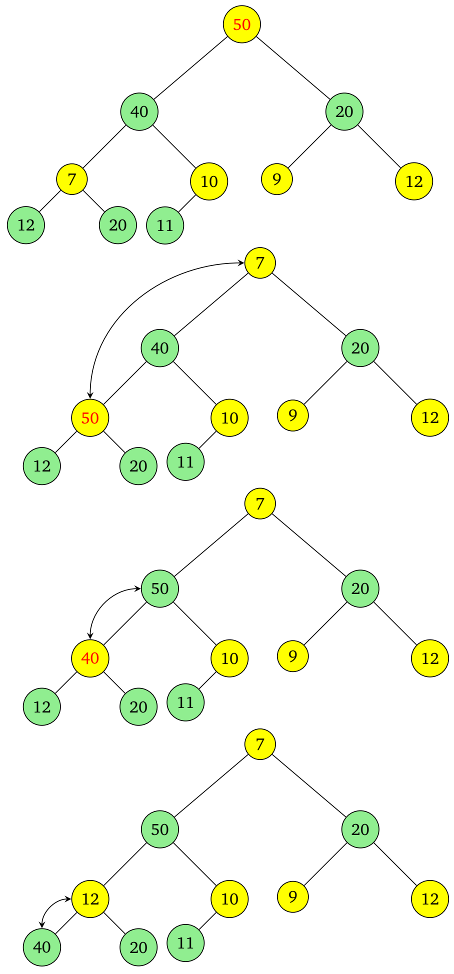


FIGURE 4.2: Trickle down operation starting from a min-level

that the element in position i is larger than its parent, thus destroying the heap property. We check this and swap the elements accordingly. At this point a node that was originally in the max-level may reach a min-level. So, we recursively apply the trickle down procedure starting from that level. The sequence of operations are illustrated in Figure 4.2.

Exercise 4.3. Write down the pseudo-code for TrickleDownMax(i) when the element i is in a max-level.

Exercise 4.4. Verify that the running-time for TrickleDownMin and TrickleDownMax is $O(\log n)$.

With the trickle down operation, we can obtain an algorithm that creates a min-max heap from an arbitrary array H with n elements in $O(n)$. For each position j starting from $n/2$, we apply the TrickleDownMin(j) or TrickleDownMax(j) depending on whether j is in a min-level or max-level, respectively. The same analysis as in the case of binary minheap gives the final running-time bound. The cost of ExtractMin is also $O(\log n)$ since we copy $H[n]$ to $H[1]$ and then perform TrickleDownMin(1) on the heap of size $n-1$. From the construction of the min-max heap, we know that the maximum element in the heap is in positions 2 or 3. For an ExtractMax, we copy $H[n]$ to $H[2]$ or $H[3]$ (depending on which is the largest) and then perform a TrickleDownMax from that position. Hence, we can perform both the operations in $O(\log n)$ without using any auxilliary heaps.

To describe insertions in a min-max heap, we have to describe the algorithm to bubble up an element in the heap. We will assume that the subtree rooted at i satisfies the heap property, but that this may not be true for the subtree rooted at i's predecessors owing to the value of $H[i]$. If i is a min-level, then it is checked first with its parent to see if a swap must be performed. If $k = H[i] > H[i/2]$, then we swap $H[i]$ and $H[i/2]$. Since the tree rooted at i originally satisfied the heap property and we replace $H[i]$ with a smaller value, the tree rooted at i will continue to have the heap property. But now k has moved to a max-level $i/2$ and k must be larger than all the elements in the min-levels above it. So, we can keep checking k with the max-levels above it one-by-one and bubble it up after comparing.

Similarly, if i was in a max-level, then $k = H[i]$ is compared with $H[i/2]$ and is swapped if it is smaller than $H[i/2]$. After the swap, the subtree rooted at i retains the heap property. The element k will move to a min-level after the swap, and we subsequently only have to check k with elements in the min-levels as it is bubbled up in the heap. The pseudo-code is described in Algorithm 4.2. Algorithm 4.3 gives the pseudo-code of the procedure that keeps checking the alternate levels once the swap with the parent is done. During an insertion, the new element is inserted in position $n+1$, and BubbleUp($n+1$) is called to move $H[n+1]$ to the correct position in the heap. Illustrations of the insertion procedure are shown in Figures 4.3 and 4.4.

Exercise 4.5. Write down the pseudo-code for BubbleUpMin.

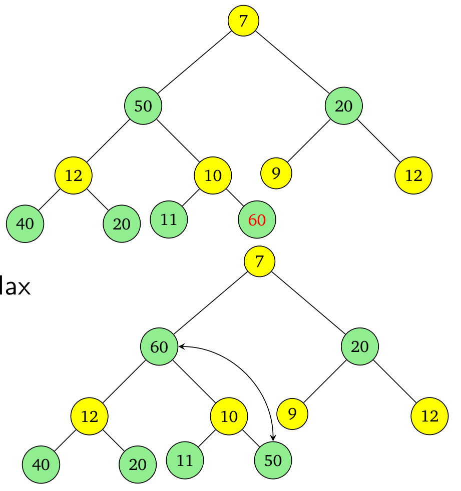


Figure 4.3: Insertion of 60 in the min-max heap - the last level is a max-level. The element 60 is compared with its parent (10). Since it is larger, the BubbleUpMax(11) is called. This compares 60 with 50 and is swapped. Now 60 is in position 2, and the bubbling up process stops.

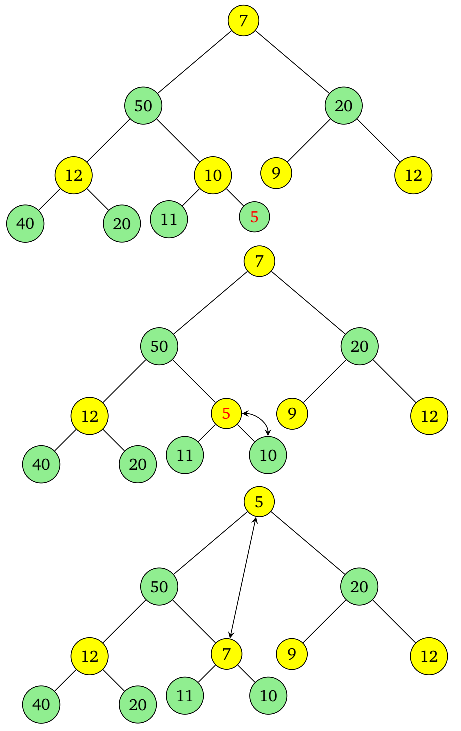


Figure 4.4: Insertion of 5 in the min-max heap - the last level is a max-level. The element 5 is compared with its parent (10). Since it is smaller, the elements are swapped and BubbleUpMin(5) is called. This will lead to a comparison of the node with 5 with its grandparen, which is the root here. The elements are swapped once more and the bubbling up process stops.

Algorithm 4.2: BubbleUp(i)

```
return if $i = 1$ //We are at the root
if $i$ is in a min-level then
    if $H[i] > H[i/2]$ then
        Swap $H[i]$ and $H[i/2]$
        BubbleUpMax($i/2$)
    else
        BubbleUpMin($i$)
else
    if $H[i] < H[i/2]$ then
        Swap $H[i]$ and $H[i/2]$
        BubbleUpMin($i/2$)
    else
        BubbleUpMax($i$)
```

Algorithm 4.3: BubbleUpMax(i)

```
return if $i = 1, 2$, or 3 //No grandparents!
if $H[i] > H[gp(i)]$ then
    Swap $H[i]$ and $H[\text{gp}(i)]$
    BubbleUpMax(gp($i$))
```

It is not hard to verify that the bubbling up operation takes $O(\log n)$ time, analogous to the trickling down procedure. Let us now look at the DecreaseKey operation. We will have multiple cases to look at depending on whether the key we are changing is in a min-level or a max-level.

An easy case is a DecreaseKey operation on a key k at position i in a min-level. Notice that decreasing a key value in a position in a min-level keeps the heap property for the subtrees rooted at i and their descendants. Thus we only need to take care of changes along the path from i to the root of the heap. Furthermore, since the decreased key value was in a min-level, we need not worry about the nodes in max-levels since they will remain to be higher than the new key value. Hence, we need to bubble up the element from position i using the BubbleUpMin procedure.

Now consider the DecreaseKey operation on a key k at position i that is in a max-level. Suppose that the new key value is $k' < k$. It is now possible that $k' < H[i/2]$, the parent of position i. The parent of i is in a min-level, and hence the subtree rooted at $i/2$ may no longer satisfy the heap property. Similarly, k' could be smaller than the grandchildren of i. This would mean that the subtree rooted at i may also not satisfy the heap property. If that is the case, then we should swap $H[i]$ and $H[i/2]$, and then bubble up from $i/2$ and trickle down from i. If $k' \geq H[i/2]$, then we would only need to trickle down from i. Filling in these details is left as an exercise.

Exercise 4.6. Describe the algorithm for DecreaseKey on a min-max heap clearly, and prove why it is correct. Write down the pseudo-code for both this operations.

4.3 Mergeable heaps

Another useful operation on a priority queue is a merge operation that combines two priority queues into one. In this section we will look at implementations of heaps that support the Merge or Meld operations. We will start with a randomized data structure that supports melding. The heap will be a binary tree, but will no longer be required to be full. The shape of the tree could be arbitrary, and hence will no longer be stored as an array. For each node in the heap, we will require additional pointers for its children.

4.3.1 Randomized mergeable heaps

We will start with a simple data structure for heaps that supports the Merge/Meld operation. The other operations of the priority queue can be performed using a constant number of Merge operations. The binary tree that stores the heap will no longer be full; in fact we will place no restrictions on its shape. The tree will always satisfy the heap property - that the key at the root is smaller than the key values of all its descendants.

For a node u, let u.left, u.right, and u.key denote the left child, right child and the key values. Let u.parent denote the parent of the node u. Since the tree has no specific structure, there could be internal nodes where u.left $=$ null and u.right contains a non-empty tree.

Let H_1 and H_2 be two minheaps with roots r_1 and r_2. Assume that r_1.key $\leq$ r_2.key; if not, we will swap the roles of H_1 and H_2. We will have r_1 as the root of the merged heap $H_1 \cup H_2$. The idea is to recursively merge H_2 with either r_1.left or r_1.right, until one of the heaps is an empty heap. At this point, we can return the root of the other heap. The choice of whether H_2 is to be merged with r_1.left or r_1.right is made uniformly with probability $1/2$ at each recursive call.

The psedudo-code for the merging two heaps is given below.

Algorithm 4.4: RandomizedMerge(H_1, H_2)

```
if $H_1 = \emptyset$ then return $r_2$
if $H_2 = \emptyset$ then return $r_1$
if $r_1.key > r_2.key$ then RandomizedMerge($H_2, H_1$)
//Now $H_1, H_2 \neq \emptyset$, and root of $H_1$ has the smaller key value
$b \leftarrow$ RandomBit()
if $b = 1$ then
    $r_1$.left $\leftarrow$ RandomizedMerge($r_1$.left, $H_2$)
    $r_1$.left.parent $\leftarrow r_1$
else
    $r_1$.right $\leftarrow$ RandomizedMerge($r_1$.right, $H_2$)
    $r_1$.right.parent $\leftarrow r_1$
return $r_1$
```

To analyze the merging process, we will look at random walks on a binary tree. A *random walk* on a binary tree starts at a root and chooses left or right at every node, with probability $1/2$. The random walk continues until the walk *falls off* the tree - i.e. we reach a node that is null. The *length* of the random walk is the number of steps taken by the random walk before it falls off, and is a random variable. The following lemma about the expected length of a random walk will give us the running-time bound for Merge.

LEMMA 4.2. *Let ℓ be the length of a random walk on a binary tree T with n nodes, starting at the root r. Then, $\mathbb{E}[\ell] = O(\log n)$.*

Proof. Firstly, add dummy nodes to every node in T that has less than 2 children, so that every node except the leaves has exactly two children. Thus every leaf in the new tree is a dummy node. The random walk ends when it has reached a dummy node. Recall that a binary tree with n nodes has at most $n+1$ leaves, and therefore in this case has at most $\Theta(n)$ dummy nodes. Let $u_1, u_2, \ldots, u_r$ be these leaf nodes, that are at depths $d_1, d_2, \ldots, d_r$.

The probability of reaching the node u_i is therefore $1/2^{d_i}$. Since the random walk ends in one of these nodes, we have $\sum_{i=1}^{r} \frac{1}{2^{d_i}} = 1$.

The expected length ℓ of the random walk can be expresses as follows.

At this point you can complete the proof by observing that the expression for the expectation is the formula for the entropy of a distribution with support $\Theta(n)$ and hence is always $O(\log n)$.

$$\begin{aligned}\mathbb{E}[\ell] &= \sum_{i=1}^{r} \frac{d_i}{2^{d_i}} = \sum_{i=1}^{r} \frac{1}{2^{d_i}} \log\left(2^{d_i}\right) \\ &\le \log\left(\sum_{i=1}^{r} \frac{2^{d_i}}{2^{d_i}}\right), \text{ because log is a concave function} \\ &= O(\log n).\end{aligned}$$

□

The insert operation is performed by merging the heap, with the heap containing just the key which is to be inserted. Similarly, ExtractMin is performed by mergin the left and right subtrees of the root. The DecreaseKey operation can similarly be done by removing the subtree rooted at the key we are changing and merging it with the original heap. Thus, all operations are performed using $O(1)$ calls to the Merge operation with some small overhead. This lets us keep the bounds on the running-time of the priority queue operations at $O(\log n)$.

4.3.2 Skew heaps

We now look at a *self-adjusting heap* that performs the merge operation deterministically, and adjusts itself to keep the costs of other heap operations small. The data structure is called self-adjusting because it does not store any auxilliary information at every node that is used in the adjustments. These data structures are thus almost as efficient as standard heaps in terms of the memory, but lose out on some of the worst-case time-bounds. Nonetheless, we will see that they give excellent amortized bounds for the operations.

Skew heaps[2] are arbitrary binary trees that satisfy the heap property. The

[2] Daniel Dominic Sleator and Robert Endre Tarjan. "Self-adjusting heaps". In: *SIAM Journal on Computing* 15.1 (1986), pp. 52–69.

main operation in the skew heap is a merge operation that recursively tries to merge two heaps H_1 and H_2. The merge operation is performed by patching up the right-most path of both the trees H_1 and H_2, and then swapping the left and right subtrees of all the nodes on this path. Before looking at the pseudocode and analyzing the algorithm, we will see an example using Figure 4.5. This figure shows two skew heaps. Note that the binary trees satisfy the min-heap property, but are not full or balanced binary trees.

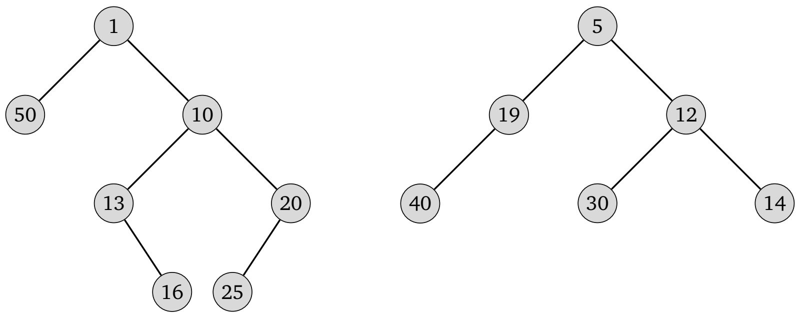


FIGURE 4.5: Merging two skew heaps H_1 and H_2.

W.l.o.g assume that the key value of the root in H_1 is smaller than the key value of the root in H_2. The merge procedure, moves the subtree rooted at the left child of the root of H_1 to the right and recursively merges the subtree rooted at the right child of H_1 with the tree H_2. The first step step in the recursive process is show in Figure 4.6.

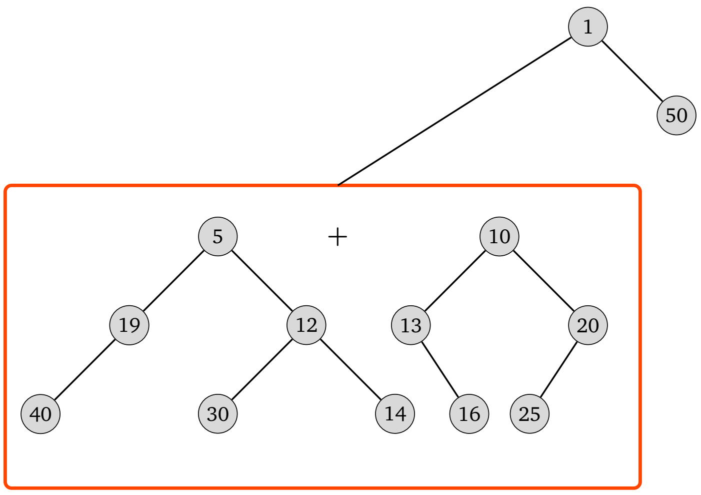


FIGURE 4.6: The first step in the recursive call for merging heaps H_1 and H_2.

The final heap formed after all the recursive calls in the merge procedure is shown in Figure 4.7.

Notice that the heap is skewed with a long leftmost path. But this means that many subsequent merges will have a shorter time complexity since each merge is done on the rightmost path followed by a switch of the left and right children along the path. This gives some intuition as to why we can expect small amortized cost for the merge operation. The pseudcode for the merge operation is given as Algorithm 4.5. You will notice that the pseudocode is similar to the randomized merging that we did earlier. The difference is that instead of choosing a random child to recursively merge, we always choose the right child and swap the left and right children as well.

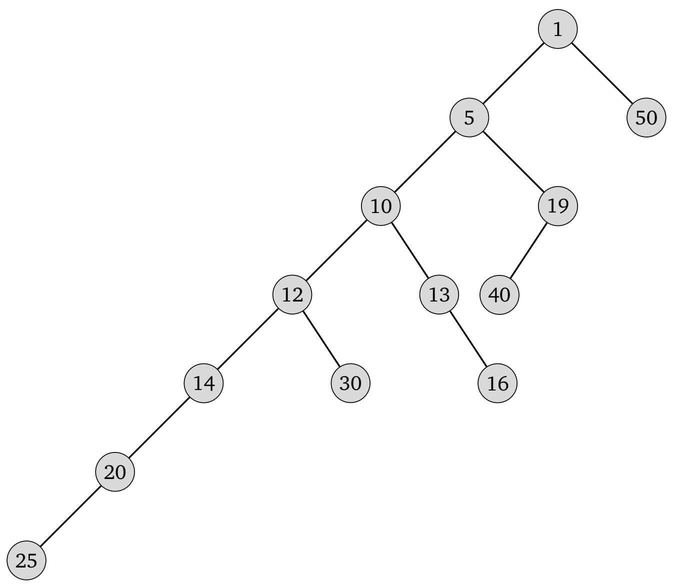


FIGURE 4.7: The final skew heap formed by the merging of H_1 and H_2.

Algorithm 4.5: MERGE(r_1, r_2)

```
//$r_1$ and $r_2$ are the roots of the two heaps
**if** $r_1$ = null **then return** $r_2$
**if** $r_2$ = null **then return** $r_1$
**if** $r_2$.key $<$ $r_1$.key **then return** MERGE$(r_2, r_1)$
temp $\leftarrow$ $r_1$.rchild
$r_1$.rchild $\leftarrow$ $r_1$.lchild
$r_1$.lchild $\leftarrow$ MERGE$(r_2$, temp$)$
**return** $r_1$
```

We will now analyze the amortized cost of the merge operation using the potential method. To that end, we need a few definitions of heavy and light nodes. We will denote by $n(u)$, the number of nodes in the subtree rooted at u (including u). A node u is said to be *heavy* if $n(u) > n(u.\text{parent})/2$. Otherwise u is said to be *light*. From the definition, it is clear that each node u has at most one heavy child.

LEMMA 4.3. *In any binary tree, for every node u and a descendent v of u, there are at most* $\log n$ *many light nodes in the path from u to v.*

Proof. For every light node u' in the path $n(u') \leq n(u'.\text{parent})/2$. Thus, if there are k light nodes in the path from u to v, $n(v) \leq n(u)/2^k$. Hence we have

$$k \leq \log\left(\frac{n(u)}{n(v)}\right).$$

Since $n(v) \geq 1$ and $n(u) \leq n$, we have $k \leq \log n$. □

We will call a node u *right-heavy* if it is a heavy node and is the right child of u.parent. Let T_i be the skew heap after the i^{th} operation, we will define the potential as

$$\Phi(i) = |\{u \in T_i \mid u \text{ is right-heavy}\}|.$$

The intuition behind this definition of the potential function is that the rightmost path that is being merged between the two trees contain at most $\log n$ light nodes. All the remaining nodes that are visited during MERGE are heavy, and they become left children (due to the swap). Thus, they can pay from their potential towards the merging. In effect, the MERGE operation needs to pay only for the light nodes, and the potential from the heavy nodes is used for the rest.

Consider the merging of two heaps H_1 and H_2 such that H_1 has a right-most path of length n_1 and H_2 has a righ-most of length n_2. The actual cost of the merge operation as described in Algorithm 4.5 is $n_1 + n_2$. The total number of light nodes in this right-most path across the two heaps is at most $\log(n_1) + \log(n_2) \leq 2\log n - 1$, where $n = n_1 + n_2$. Let s_1 and s_2 be the number of heavy nodes in the right-most path of H_1 and H_2, respectively, that were visited during the merge operation. Since all these nodes are in the right-most path, they are right-heavy. But after the MERGE process, they becomes left children of their parents. Similarly, the siblings of light nodes on the right-most path are potentially heavy. Thus, after the MERGE, they become right children of their parents and can become right-heavy. Since there are at $\log n$ many light nodes in the rightmost path, we have

$$\Phi(i-1) - \Phi(i) \geq s_1 + s_2 - \log n.$$

The amortized cost of the i^{th} merge operation $\hat{c}_i$ is given by

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(i) - \Phi(i-1)\\ &\leq (s_1 + s_2 + 2\log n - 1) + (\log n - s_1 - s_2)\\ &= 3\log n - 1.\end{aligned}$$

The Insert operation is performed by creating a heap with a single element and performing a merge operation. The GetMin operation takes $O(1)$ time since the minimum is at the root of the tree. The ExtractMin operation is done by removing the root and merging the subtrees rooted at the left and right children of the root. The DecreaseKey operation can be performed by splicing the tree rooted at the node whose value is being reduced, and then merging it back into the original tree.

THEOREM 4.4. *A skew heap implements Insert, ExtractMin, and DecreaseKey operations in $O(\log n)$ amortized time, and GetMin in $O(1)$ worst-case time.*

A slight variant of the merge procedure, where the merging on the right-most path is performed bottom-up gives amortized bounds for $O(1)$ for Insert and ExtractMin. But even this implementation gives $O(\log n)$ amortized bound for DecreaseKey operation. Many graph algorithms that use priority queues require DecreaseKey operations. We will now see an implementation of mergeable heaps that perform DecreaseKey in $O(1)$ amortized cost. These will no longer be self-adjusting as we will need to store auxilliary information per node.

4.3.3 Binomial heaps

The variant of mergeable that we are going to see now are called binomial heaps.[3] These data structures are slightly clunkier than before as each node to store multiple pointers and auxilliary information.

[3] Jean Vuillemin. "A data structure for manipulating priority queues". In: *Communications of the ACM* 21.4 (1978), pp. 309–315.

A binomial heap is a collection of *binomial trees*, each of which satisfies the heap property - the root of the tree contains the smallest key among all the keys in that tree. A binomial tree of order k, denoted by B_k, is a tree with 2^k nodes, obtained by pointing the root of a binomial tree of order $k-1$ to the root of another binomial tree of order $k-1$. See Figure 4.8 for binomial trees of orders 0 to 3.

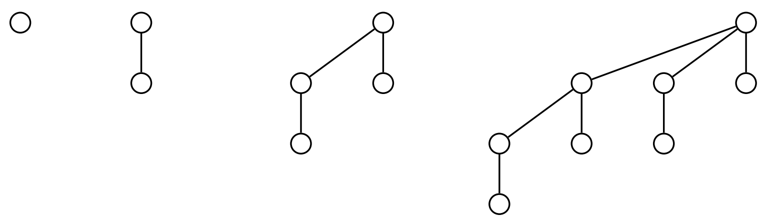


FIGURE 4.8: Binomial trees of order $k = 0, 1, 2, 3$.

The name binomial tree comes from the following property of such trees.

LEMMA 4.5. *The number of nodes in level ℓ of B_k is $\binom{k}{\ell}$.*

Proof. The proof follows from induction on k. You can verify that the statement holds for B_0 and B_1. Consider an arbitrary level ℓ of B_k. Since B_k was formed by taking the union of two B_{k-1}s, the nodes in level ℓ are all the nodes in level $\ell - 1$ in one of the B_{k-1}s and all the nodes in level ℓ of the other B_{k-1}. Thus the number of nodes in level ℓ of B_k is $\binom{k-1}{\ell-1} + \binom{k-1}{\ell}$ (by induction hypothesis). Hence, the number of nodes at level ℓ of B_k is $\binom{k}{\ell}$. □

A binomial heap on n elements is a collection of r trees, where r is the number of ones in the binary representation of n. For instance, if $n = 13$, then $r = 4$ and the binary representation is 1101. The binomial heap consists of the binomial trees B_0, B_2, and B_3. Each of B_0, B_2, and B_3 will satisfy the heap property and the smallest element in the binomial heap will be one of the roots of B_0, B_2, or B_3.

A binomial heap is implemented by connecting all the roots of its binomial trees via a doubly linked list. Each node in a binomial tree has a child pointer that points to the child with the highest order, a pointer to its sibling of the next lower order and next higher order - i.e. the children of a node are connected via a doubly linked list. It also has a parent pointer that will be useful during a DecreaseKey operation. We will also maintain a pointer to the node with the lowest key value so that GetMin can be implements in $O(1)$ time.

The key operation that we will look at is the Merge. The other operations of the priority queue are implemented using multiple calls to Merge.

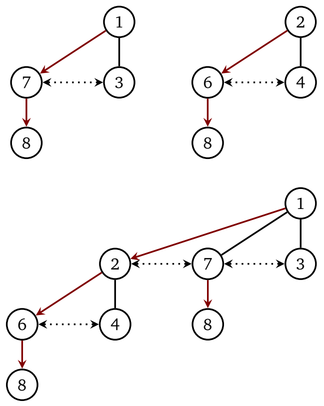


FIGURE 4.9: The merging of two binomial trees of order 2. The red edges denote the pointer to the child of highest order. The dotted edges denote

Let's start with how we will merge two binomial trees T_1 and T_2, both of order k. This is illustrated in Figure 4.9. Let r_1 and r_2 be the roots of T_1 and T_2, respectively, and let r_1.key $< r_2$.key. We will first update the sibling pointer of r_2 to the node pointed to by the current child pointer of r_1, and similarly update the sibling pointer of that node. Next, we will update the child pointer of r_1 to point to r_2. Thus, we can perform the merging of two binomial trees of the same order in $O(1)$-time.

The Merge operation of a binomial heap is done by repeated merging of the binomial trees that constitute these heaps. Consider the following two heaps that we will call H_1 and H_2.

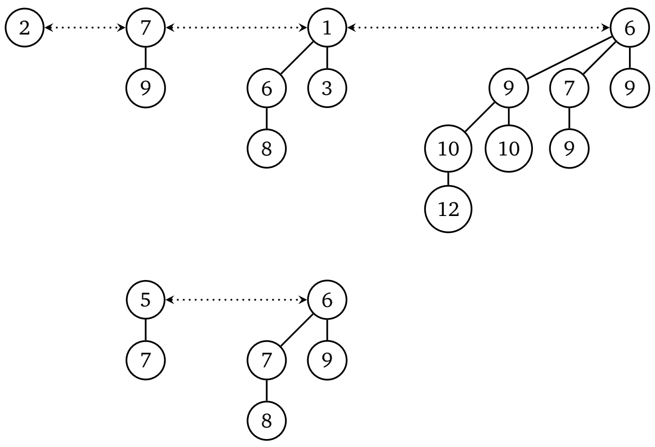


FIGURE 4.10: Merging two heaps. The first heap has $n = 15$ with binary representation 1111. The second heap has $n = 6$ with binary represenation 0110. The merging of these two heaps will create a new heap with 21 elements. Since the binary representation of 21 is 10101, the merged heap will have B_4, B_2 and B_0.

The idea is to perform binary addition, where the addition operation corresponds to merging of two binomial trees. Starting from $i = 0$, the corresponding binomial trees B_i in the two heaps are merged. If only one of the heaps has the binomial tree B_i, then it is kept as it is in the merged heap. If both the heaps have a binomial tree of order i, they are merged to form a binomial tree of order $i + 1$. This is like the carry operation in binary addition.

In the example illustrated above, there is only one tree of order 0 and it is retained as it is in the merged heap. There are binomial trees of order 1 in both heaps. These are merged to create a binomial tree of order 2 in the merged heap. Since there are two binomial trees of order 2 in the two heaps, they are merged to create a binomial tree of order 3. Now there is a binomial tree of order 3 in H_1. This is merged with the new tree of order 3 that was created by the earlier merging step, and we get a tree of order 4. Thus, the final heap has a B_0, B_2, and B_4. The final merged binomial heap is shown in the figure below.

Since a binomial heap of size n has $\lfloor \log n \rfloor$ binomial trees constituting it, and the time for merging two binomial tree of same order is $O(1)$, the total running time for mergin/melding two binomial heaps of size n is at most $O(\log n)$. The other heap operations can be performed using the Merge operation. For instance, Insert operation on a heap H by a key k can be done by thinking of the key k as a heap with a single element and merging it with H. Since we maintain a pointer to the smallest element, the GetMin operation can be performed in $O(1)$-time.

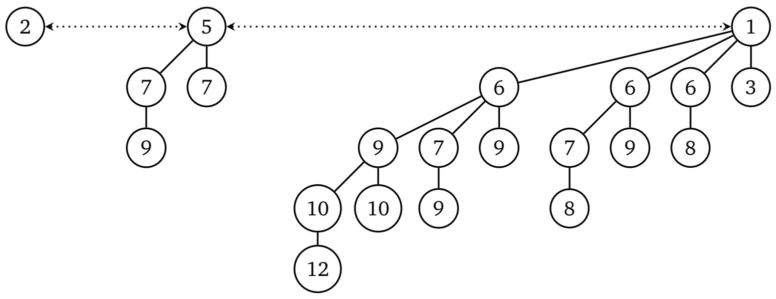


FIGURE 4.11: Final merged heap obtained from H_1 and H_2.

For the ExtractMin operation, we will first remove the binomial tree containing the smallest element from the heap. Suppose that is was of order k. Then the children of the root themselves are a binomial heap of size 2^{k-1}. We will then merge this heap with the former. We will then traverse the root list to maintain the smallest element in the merge list. This overhead which is $O(\log n)$ helps us perform GetMin in $O(1)$-time. For the heap in Figure 4.11, if we perform ExtractMin, then the node with value 1 is removed, and its children form a new heap. This new heap is merged to get the final heap after ExtractMin. This is illustrated in Figure 4.12.

The DecreaseKey operation can be performed by the standard bubbling up operation on the binomial tree that contains the key that is decreased. In a binomial heap on n keys, the largest binomial tree has size at most $\log n$, and height at most $\log n$. Thus the DecreaseKey operation can also be perform in time $O(\log n)$.

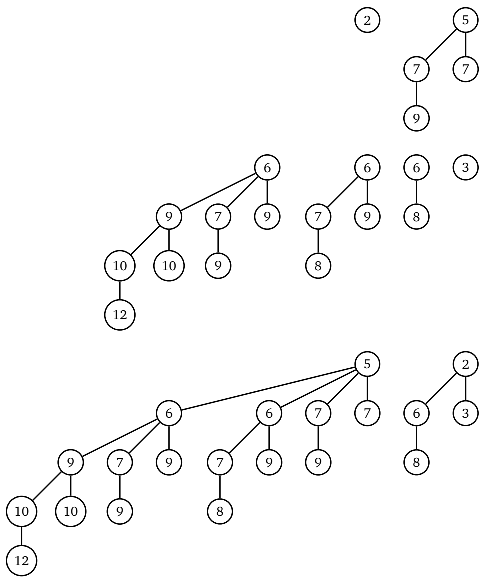


FIGURE 4.12: ExtractMin gives two binomial heaps which are then merged.

4.3.3.1 Amortized complexity of insertions

We will now show that the amortized complexity of Insert in a binomial heap is $O(1)$, even though the worst-case complexity is $O(\log n)$. We will analyze the amortized complexity using the potential method. To get an intuition about the choice of the potential function, recall that during an Insert operation, the basic operation performed on the heap is the merging of binomial trees. The cost of merging two binomial trees of the same order is $O(1)$. With this in mind, let us define $\Phi(H_i)$ to be the number of binomial trees in the heap H_i after the i^{th} operation.

Suppose that $\Phi(H_{i-1}) = k$. Let ℓ be the smallest value such that H_i has binomial trees of order $i \leq \ell$ and $B_{\ell+1}$ is not present. An insertion operation on H_i will create a heap H_{i+1} that do not contain any of the binomial trees of order at most ℓ, and contains the binomial tree of order $\ell + 1$. Thus the actual cost of the insertion is $t_i = \ell$. Also, $\Phi(H_i) = k - \ell + 1$. Thus the amortized cost of an insertion is $\widehat{t_i} = t_i + \Phi(H_i) - \Phi(H_{i-1}) = O(1)$.

4.3.3.2 Lazy binomial heaps

We will now see how to make the Merge operation *lazy*. We will see that this makes both Insert and Merge run in $O(1)$-time worst-case. Unfortunately, the ExtractMin will become $O(n)$ in the worst-case. We will show that we can also manage to do it $O(\log n)$ amortized cost.

The idea here is that we will forgo the property that there can be at most one binomial tree of a particular order in the heap. This will let us perform Insert and Merge in $O(1)$-time. We will then clean up the binomial heap after an ExtractMin operation. The Insert and Merge procedure is to merely add the new binomial trees into the existing list of binomial trees. This amounts to updating a couple of pointers and can be done in $O(1)$-time. For instance, if we perform n insertions starting from the empty heap, the lazy binomial heap will be just be a linked list of n elements.

The clean-up that is done during ExtractMin is as follows: The first step is to delete the minimum element, and merge its children into the main heap. The pointer to the smallest element can be maintained in $O(1)$-time during Insert, Merge and DecreaseKey operations. If there are n elements in this lazy heap, maintain $\ell = \log n$ pointers $P_1, P_2, \ldots, P_\ell$ such that each P_i is either null or points to the root of a binomial tree of order i. Initially all the P_i are null. While traversing the root list of the lazy heap, for each tree in the heap of order i, if P_i is null then point P_i to the root of the tree. Otherwise, merge the tree pointed to by P_i and the current tree of order i and check with the tree pointed to by P_{i+1}. Continue this process until we find an empty P_j, $j > i$. At the end of this process, we have a binomial heap whose trees are pointed to by $P_1, P_2, \ldots, P_\ell$.

Maintain a global pointer for each heap, and update it by comparing. Verify that this is indeed possible.

We will analyze the amortized cost of ExtractMin using the accounting method. For each insertion, we will give an additional $O(1)$ credit to the element that is inserted. This will be used for a later merging operation with another binomial tree of the same order. During ExtractMin, all the children of the minimum element are added in the root list of the heap, and each of these elements are $O(1)$ units of credit. Thus, we spend $O(\log n)$ time for the initial deletion step of ExtractMin. We would now like to show that we have enough credit to take care of the clean-up operation.

After the deletion operation, we have a heap that consists of binomial trees of order up to $\log n$. Furthermore, each of the roots of these trees have $O(1)$ units of credit that they can use for merging. For merging two tree T_1 and T_2 of order i such that root of T_1 becomes the new root, we will use the credit available in the root of T_2 to perform this $O(1)$ operation. After a node becomes a child of the root, it is never involved in any of the further merge operations of that tree. Thus, we can perform the clean-up operation to get a binomial heap such that the roots of each of its constituent binomial trees have $O(1)$ units of credits with itself for future operations.

4.3.4 Fibonacci heaps

We will now modify the lazy binomial heaps in the previous section to obtain a new data structure that will also let us perform the DecreaseKey operation in amortized $O(1)$-time. The DecreaseKey operation is important in various shortest path algorithms, as we will see later, and using this data structure will give the best asymptotic running-time bounds for those algorithms.

A Fibonacci heap[4] consists of a collection of trees that satisfy the heap property. The trees need no longer be binomial trees. The order of a tree will now be defined as the number of children of its root. Merging two such

[4] Michael L Fredman and Robert Endre Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms”. In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

trees of order k will create a new tree of order $k + 1$. This is similar to the merging of binomial trees, and is an $O(1)$ operation. The ExtractMin operation remains as in the case of lazy binomial heaps. We will change the DecreaseKey operation to a lazy version, and obtain the $O(1)$-amortized running-time bound.

Suppose that we have a DecreaseKey operation on a node in a tree. If the operation does not destroy the heap propery, then we can proceed with the operation and stop in $O(1)$. On the other hand, if the decrease of the key leads to the violation of the heap property, we will splice that subtree from its parent and add it as a new tree in the heap. In fact, this would let us perform DecreaseKey in $O(1)$-time.

The problem with this lazy method is that we might end up with trees of order k that contains only $k + 1$ vertices. The requirement that the largest order for any tree in a heap with n nodes should be $\log n$ will be violated. This was what made the clean-up during ExtractMin work in $O(\log n)$-amortized time. To avoid this issue, we will perform what is known as a *cascading cut*, that splices a node if at least two of its children has been spliced from it due to DecreaseKey operations.

Now, each node u stores a boolean value u.mark that is set if one of its children has been spliced. If a marked node has another child spliced from it, then u splices itself from its parent and adds itself to the root list and set u.mark to false. The parent of u may continue this process upwards, and it stops when an unmarked node or the root is reached.

There are two things that we need to prove here.

1. Even though the cuts are cascading, the amortized cost of DecreaseKey is $O(1)$.
2. The maximum degree of any tree in the heap is $O(\log n)$ - this would be required for the ExtractMin to have an amortized running time of $O(\log n)$.

The first property is proved in the following lemma.

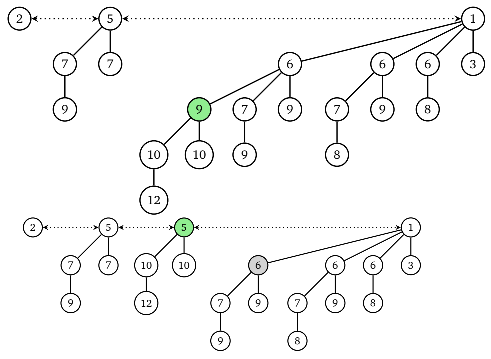


FIGURE 4.13: When the DecreaseKey operation is performed on the green node and its value is reduced to 5, the node is spliced and added to the root list. It's parent node (with value 6) is marked. The tree of order 4 (rooted at the element 1) in the heap is now no longer a binomial tree.

LEMMA 4.6. *The amortized complexity of DecreaseKey operation on a Fibonacci heap is $O(1)$.*

Proof. We will once again use the accounting method. First, observe that splicing an element and adding it into the root list is an $O(1)$ operation. We will additionally pay $O(1)$ units of credit to this node which has become the root (for further clean-up operations), and pay $O(1)$ credit to its parent if it is unmarked. The credit given to the unmarked node is for the cost that it has to bear during a cascading cut. Hence, only the node whose value is decreased and the final unmarked node on the trail of the cascading cut need to be paid for. The costs of removing marked nodes in the path and adding it into the root list has been paid for when the node was marked. Therefore, the amortized cost of DecreaseKey is $O(1)$. □

The proof of the second property will also justify the name Fibonacci heaps that we gave for this lazier version of mergeable heaps. To that end, we start with the following lemma about the nodes in the trees constituting a Fibonacci heap.

LEMMA 4.7. *Let u be any node of order r in a Fibonacci heap, and let $v_1, v_2, \ldots, v_r$ be its r children in the order in which they were merged with u (v_1 being the first). Then for every $i \geq 2$, order of v_i is at least $i-2$.*

Proof. Consider the step in which v_i became a child of u. At this point u must have had at leat $i-1$ children. The reason v_i became a child of u was that the order of u and order of v_i at that point was the same. Therefore, at the time of the merging step when v_i became a child of u, the order of v_i was at least $i-1$. If v_i remains as a child of u, then it would have lost at most one child by the splicing operation during DecreaseKey. Hence the order of v_i is at least $i-2$. □

To obtain an upper bound on the order of the trees in a Fibonacci heap, we will use a lower bound for the minimum size of a tree of order k in a Fibonacci heap.

LEMMA 4.8. *Let s_k denote the minimum size of a tree of order k in a Fibonacci heap. Then $s_k \geq \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.*

Proof. First note that $s_0 = 1$ and $s_1 = 2$ since a node with zero children has size 1 and a node with 1 child has size 2. We will prove that s_k satisfies the following recurrence for $k \geq 2$.

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}.$$

Let u be a node with k children $v_1, v_2, \ldots, v_k$, ordered in the sequence in which they became u's children. Let r_i denote the order of the node v_i. For every $i \geq 2$, $r_i \geq i-2$ (from Lemma 4.7), and $r_1 \geq 0$ (trivially). The function s_k is monotonically increasing in k, and hence $s_{r_i} \geq s_{i-2}$. Since $r_1 \geq 0$, we have $s_{r_1} \geq 1$. The recurrence follows by adding all up the nodes in each of the subtrees together with root u.

For obtain the lower bound on s_k from the recurrence, let's recall the Fibonacci numbers defined by the recurrence $F_k = F_{k-1} + F_{k-2}$ for $k \geq 0$ with the base cases $F_0 = 0$ and $F_1 = 1$. We need a few properties of F_k that we state below, whose proofs follow by simple induction.

CLAIM 4.9. *For every $k \geq 0$, $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.*

CLAIM 4.10. *For every $k \geq 0$, $F_{k+2} \geq \phi^k$ where $\phi = \frac{1+\sqrt{5}}{2}$.*

Using the claims stated above, we will prove that $s_k \geq F_{k+2}$ to complete the proof of the lemma. This will also be proved using induction on k. Clearly $s_0 = 1 \geq F_2 = 1$ and $s_1 = 2 \geq F_3 = 2$ proving the base case of the induction. By the induction hypothesis, $s_i \geq F_{i+2}$ for all $i < k$. Thus we can rewrite the recurrence involving s_k as follows.

$$s_k \geq 2 + \sum_{i=2}^{k} F_i \geq 1 + \sum_{i=0}^{k} F_i = F_{k+2}.$$

□

EXERCISE 4.7. Prove Claims 4.9 and 4.10.

As a simple corollary of the lemma we can see that if the maximum order of any tree in a Fibonacci heap with n elements is k, then $n \geq s_k \geq \phi^k$. Therefore $k = O(\log n)$. This means that the amortized complexity of ExtractMin is $O(\log n)$ and follows from the analysis of the lazy binomial heap that we did in the last section.

4.3.4.1 Dijkstra's shortest path algorithm

We will briefly describe how using Fibonacci heaps in Dijkstra's algorithm gives the asymptotically tightest running-time bound of $O(|E| + |V| \log |V|)$ in the worst-case. Let us start by setting up the notation for the shortest path problem.

Given a weighted graph $G(V, E, w)$ where $w : E \to \mathbb{R}^+$ and a source vertex $s \in V$, we wish to compute the shortest paths from s to all the vertices in G. Dijkstra's algorithm maintains a set $S \subseteq V$ that contains all the vertices whose shortest paths from s have been computed, and grows S until $S = V$. At every step the algorithm maintains $d(v)$ which is an upper bound on the actual shortest distance $d^*(v)$ from s to v.

The invariant that is maintained by the algorithm will be that for all the vertices in $u \in S$, $d(u) = d^*(u)$. Initially $S = \{s\}$ contains only the source vertex s. At this point $d(s) = 0$ and $d(v) = \infty$ for all $v \in V - \{s\}$. To see how the invariant is maintained in the subsequent steps, we prove the following lemma.

LEMMA 4.11. *Let $S \subseteq V$ be such that for every $u \in S$, $d(u) = d^*(u)$. For every $v \in V - S$, set $d(v) = \min_{u \in S}\{d(u) + w(u, v)\}$. Consider the vertex $x \in V - S$ such that $x = \arg\min_{v \in V - S}\{d(v)\}$. Then $d(x) = d^*(x)$.*

Proof. The proof will crucially use the fact that $w(e) \geq 0$ for every edge $e \in E$. We will prove by contradiction. Suppose that $d(x) > d^*(x)$.

Consider the shortest path from s to x. Let $(v, x') \in E$ be the first edge in that path such that $v \in S$ and $x' \in V - S$. Let $d(x', x)$ refer to the shortest path from x to x'. Then $d^*(x) = d(v) + w(v, x') + d(x', x)$.

For x, let $u \in S$ be the vertex for which $d(u) + w(u, x)$ is minimized. From the choice of x, we know that $d(u) + w(u, x) \leq d(v) + w(v, x')$. But this would mean that

$$d(x) > d^*(x) = d(v) + w(v, x') + d(x', x) \geq d(u) + w(u, x) = d(x),$$

since $d(x', x) \geq 0$. Hence it must be the case that $d(x) \leq d^*(x)$. Since $d^*(x)$ is the length of the shortest path from s to x, we have $d(x) = d^*(x)$. □

The underlying algorithm iteratively builds the set starting from $\{s\}$ all the way to V. Lemma 4.11 gives a way to find the next element to be added to S at each step of the algorithm. We will now discuss the ideas involved in making this algorithm efficient using suitable data structures which will help in obtaining the x given by Lemma 4.11 at every step.

We need a data structure to maintain the set $V-S$ from which the vertex x that satisfies the condition of Lemma 4.11 can be easily extracted. Priority queues seem a natural choice where the nodes in the queue correspond to the vertices in V, and the priorities correspond to the $d(v)$ values. Obtaining the x in Lemma 4.11 will correspond to an ExtractMin operation on the priority queue.

The priority queue is initially constructed by adding all vertices in V, with $d(s) = 0$ and $d(v) = \infty$ for all $v \in V - \{s\}$. The set $S = \emptyset$ initially. The first ExtractMin operation will extract s from the priority queue and update the value of $d(v)$ for the other vertices. Notice that only the vertices that are neighbors of vertices in S are updated. At an intermediate stage, when a new vertex u is extracted from the priority queue and added to S, we need to update the $d(v)$ values for only those vertices v that are neighbors of u. The other values remain unchanged. This operation performed by the algorithm is a *relaxation* of the edges going out of u.

$$\forall v \in N(u) \cap (V-S): \quad d(v) \leftarrow \min\{d(v), d(u)+w(u,v)\}$$

This operation of modifying the values of $d(v)$ for every $v \in N(u)$ is performed as a DecreaseKey operation on the priority queue. The nodes in the priority queue that are neighbors of u will precisely be $N(u) \cap (V-S)$. There are two key points to note here that will determine the running time of the algorithm.

1. Each vertex is inserted into the priority queue at the beginning of the algorithm and extracted exactly once. Thus, the total time for these operations is $O(|V| \log |V|)$ in the worst-case.

2. A relaxation of an edge is performed exactly once when one of its end points is extracted from the heap and added to S. Thus the total number of DecreaseKey operations performed during the entire run of the algorithm is $|E|$. If we use a Fibonacci heap for the priority queue, the the running time for all these DecreaseKey operations is $O(|E|)$.

Hence an implementation of Dijkstra's algorithm using Fibonacci heaps to maintain the set $V-S$ will give an overall running time of $O(|E| + |V| \log |V|)$.

5 Disjoint sets

We now look at data structures to represent sets. A set consists of elements from a universe $\mathscr{U}$ which is not necessarily ordered. The basic operations that we want to support on the ADT are the following.

- $\mathsf{MakeSet}(u)$ - create the singleton set $\{u\}$.
- $\mathsf{Find}(u)$ - return the id of the set containing the element u.
- $\mathsf{Union}(u, v)$ - return the id of the set formed by taking the union of the sets containing u and v.

Implementations of this data structure has applications in spanning tree algorithms, especially Kruskal's algorithm, and dynamic graph algorithms. We will look at ways to implement this ADT before looking at applications.

5.1 List-based implementation

A simple way to implement disjoint sets would be to use a linked list for each of the sets. For a list L, the element at the head of the list L.head will be the identifier of the set. Thus, adding an element in the set will be an $O(1)$ operation in the worst-case.

The union operation $\mathsf{Union}(u, v)$ will first need the Find operation on u and v to find the sets that they belong to. If they belong to the set, then the union operation does not change the sets. Thus the complexity of union is at least as large as the complexity of the Find operation. For a linked list, the Find operation can potentially take $O(n)$-time since we may have to traverse the entire linked list.

Now for each element u in the set L, we could add additional information - u.setid that points L.head. This way we can perform Find in $O(1)$ time. But this creates a new problem: Each time we perform a union, we need to modify u.setid for each element in the set that we are merging.

Exercise 5.1. Write the pseudocode for the two version of disjoint sets using linked lists.

One heuristic that we can think of at this stage is to attach the smaller set to the larger set during a union operation. This way the number of updates of the head pointer is the minimum of the two set sizes. This will still not avoid the worst-case $O(n)$ complexity, but we will see that the amortized cost of the operations is better. We will use L.size to store the size of the set contained in the list L. For an operation Union(u, v), we will first check if u.setid $= v$.setid. If they are different, we will compare L_1.size and L_2.size, where L_1 and L_2 are the lists containing u and v, respectively. If L_1.size $>$ L_2.size, then we add the list containing v into the list containing u, and update the head pointers of all the elements in L_2. This would take time $O(\min\{L_1.\text{size}, L_2.\text{size}\})$.

Instead of single operation (which can potentially cost $O(n)$-time), let us look at a sequence of m operations and n MakeSet operations. The m operations include Find and Union operations, and the number of elements in the disjoint sets is at most n over the entire sequence of operations. Since the Find operation takes $O(1)$ time, the total time taken over all the (at most) m Find operations is $O(m)$.

To analyze the Union operation, we will count the number of times the head pointer is changed for each element. The sum over all the elements will be the total time taken over all the Union operations. After each operation, the size of the smaller set at least doubles. Since we only update the head information for the elements of the smaller set, the total number of times the head information is updated for an arbitrary element u is $\log n$. Since there are at most n elements in the sets (there are only n MakeSet operations in total), the total time for all the Find operations is at most $O(n \log n)$. This gives the following statement about the complexity of operations when disjoint sets are implemented using linked lists.

LEMMA 5.1. *If a disjoint set data structure is implemented using linked lists, then the total cost of n MakeSet operations and m operations of Find and Union is at most $O(m + n \log n)$.*

5.2 Tree-based implementation

An alternate way to represent sets is to use trees instead of lists. The nodes in the tree correspond to the elements of the set, and each node has a pointer to its parent. The id of the root node is the id of the set. Since the number of childrenfor each node is unbounded, children pointers are not maintained in most implementations.

The MakeSet operation creates a single node tree with parent pointer pointing to itself. This takes $O(1)$-time. The Find operation follows the parent pointers until the root, and returns the id of the root. The union operation Union(u, v) performs Find(u) and Find(v), and connects the parent of the root of one of the trees to point to the root of the other tree. The worst-case running time of Find and Union depends on the depth of tree representing the set. This can be as bad as $O(n)$ if the Union operation is performed poorly - for instance, the tree might end up being a linked list due to the unions. We will see two ways to implement the union operations - one that will give good worst-case bounds, and another that will give good amortized bounds.

We will identitfy the root of the tree as the node u such that u.parent $= u$.

5.2.1 Union-by-rank

We will start be defining the *rank* of each node u in the tree. This will be done inductively.

$$\mathsf{rank}(u) = \begin{cases} 0 & \text{if } \{u\} \text{ is a singleton set} \\ 1 + \max_{i \in \{1,2,\ldots,k\}}\{\mathsf{rank}(v_i)\} & \text{if } v_1, v_2, \ldots, v_k \text{ are the children of } u \end{cases}$$

In other words, the rank of a node u is the height of the node in the tree. We will not refer to it as height because in the next section, we will modify the algorithm in a way such that the rank will no longer be equivalent to the height. Since the worst-case time complexity for $\mathsf{Find}(u)$ will the depth of the tree containing u, we would like to keep the rank of the trees to be as small as possible. To do this, we will add the root node with lower rank to the root node with the larger rank.

Suppose that $r_1 = \mathsf{Find}(u)$ and $r_2 = \mathsf{Find}(v)$ are the roots of the trees containing u and v. If $\mathsf{rank}(r_1) < \mathsf{rank}(r_2)$, then we set $r_1.\mathsf{parent} = r_2$. We do the opposite if $\mathsf{rank}(r_1) > \mathsf{rank}(r_2)$. On the other hand, if $\mathsf{rank}(r_1) = \mathsf{rank}(r_2)$, then we arbitrarily map one to the other - say $r_1.\mathsf{parent} = r_2$ and we will increment $\mathsf{rank}(r_2)$ by 1. Note that the trees constructed using union-by-rank need not necessarily be binary trees. The branching factor could be arbitrarily large. See Figure 5.1 for an example.

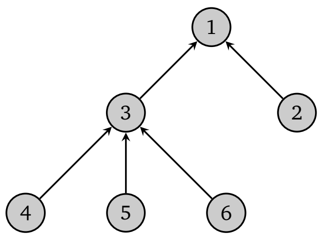


Figure 5.1: Union-by-rank - Here we have six MakeSet operations on the elements 1 to 6, followed by $\mathsf{Union}(1,2)$, $\mathsf{Union}(3,6)$, $\mathsf{Union}(4,6)$, $\mathsf{Union}(3,5)$, $\mathsf{Union}(2,3)$. Here $\mathsf{rank}(1) = 2$, $\mathsf{rank}(2) = \mathsf{rank}(3) = 1$, and $\mathsf{rank}(4) = \mathsf{rank}(5) = \mathsf{rank}(6) = 0$.

Let's observe certain properties of the trees when using union-by-rank.

Proposition 5.2. *For any node u that is not the root, $\mathsf{rank}(u) < \mathsf{rank}(u.\mathsf{parent})$.*

Lemma 5.3. *Let u be a node in the tree with rank r. Then the number of nodes in the subtree rooted at u is at least 2^r.*

Proof. Let us look at the step when u becomes a node of rank r. This must have happened due to the union of the tree rooted at u (when it had rank $r-1$) with another tree of rank $r-1$. Inductively each of the trees had at least 2^{r-1} nodes, and hence the union has at least 2^r nodes. Since a node never looses nodes from its subtree, this bound continues to hold later as well. □

Since each node with rank r has at least 2^r nodes in its subtree, and the total number of nodes is n, the following observation can be easily verified.

Proposition 5.4. *In a disjoint sets data structure with n elements in total, the number of nodes with rank r is at most $n/2^r$.*

Thus, we can see that the largest rank possible for any node in the data structure is at most $\log n$. Consequently, the path from any node to the root of the tree containing it is at most $\log n$. Hence we have the following result on the worst-case time-complexity for union-by-rank.

THEOREM 5.5. *For a tree implementation of the disjoint set data structure that uses union-by-rank, the worst-case time-complexity of* Find *and* Union *is $O(\log n)$ for a set with n elements.*

5.2.2 Union-by-rank with path compression

Path compression is an interesting heuristic that improves the amortized complexity of the Find and Union operations. The basic idea is to connect all the nodes in the path from a node u to the root directly to the root, whenever a $\mathsf{Find}(u)$ operation is executed. Thus, even though the worst-case complexity remains $O(\log n)$ (from the analysis of the earlier section), we may hope to get a better complexity for Find operations in the future. We will show that the path compression heuristic gives an amortized complexity of $O(\log^* n)$. An example of path compression is given in Figure 5.2.

In fact, the tight upper bound $\alpha(n)$, where $\alpha(\cdot)$ is the inverse Ackermann function, which is a slower growing function than the iterated logarithm.

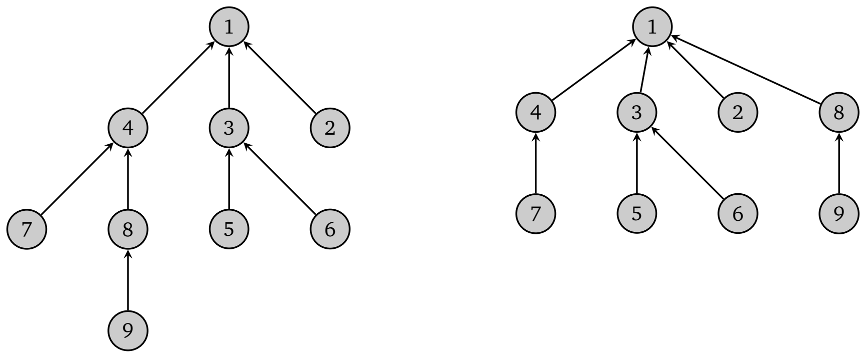


FIGURE 5.2: Path compression - the figure shows the tree corresponding to the set, and the final tree after $\mathsf{Find}(8)$ is performed on the set.

First note that with path compression, the rank of a node is no longer same as the height of the node in the tree containing it. In Figure 5.2, after the $\mathsf{Find}(8)$ operation, the height of the node 4 is 1, even though $\mathsf{rank}(4) = 2$.Nonetheless, a few of the properties from the previous section carry over for the case of path compression as well. It remains true that for all nodes u in the tree except for the root, we have $\mathsf{rank}(u) < \mathsf{rank}(u.\mathsf{parent})$. Lemma 5.3 is no longer true for all nodes in a tree, but if the root node u has rank r, then the tree contains at least 2^r nodes. The reason the statement does not hold for internal nodes is that child nodes may now be directly connected to the root following a path compression operation. Note that once a node becomes an internal node, its rank remains unchanged. Thus it must be the case that the number of nodes of rank r is at most $n/2^r$, just like in the case when we performed union-by-rank without path compression.

Amortized analysis of path compression

We will now show that the amortized complexity of Find and Union operations is almost a constant - it will be $O(\log^* n)$, where $\log^*$ is the iterated logarithm function. During each Find operation on a node u, the value of

u.parent increases. The largest rank for any node in the tree is $\log n$, as we have already seen earlier. The Find operations start becoming costly once a node stops being the root of a tree. We will use the accounting method to assign a certain amount of credit to a node once it becomes an internal node (due to a Union operation), and show that there will be sufficient credits available to perform Find operations cost-effectively. We will not worry about the Union operation since their cost is bounded by the cost of the Find operations.

Recall that each Find operation involves a traversal of a path from a node u to the root of the tree containing u. Our accounting strategy will pay for some of the edges in the path, and the remaining will be counted as the amount paid in that particular step. Each node u of rank k will be credited with 2^k units when it becomes an internal node during a Union operation. This will be sufficient to pay for traversing the edge from u to u.parent at most 2^k times. After one traversal from u to u.parent happens during a Find operation, the new parent has a larger rank than the old parent of u. Thus, after paying the 2^k units we can be sure that u.parent has rank at least 2^k. Hence, the cost that Find has to pay outside of the amount which has been credited will be the number of edges from nodes of rank k to nodes of rank 2^k. The number of such edges is in fact at most $\log^* n$. Let us carefully finish this analysis.

First, we will bound the total amount that we pay as credit across all operations. This must be added to the final cost of the operations. We will first divide the ranks into buckets with a bucket consisting of ranks from $\{k+1, \ldots, 2^k\}$ starting with $k = 0$. For instance, the buckets would be

$$\{1\}, \{2\}, \{3,4\}, \{5,6,\ldots,16\}, \{17,18,\ldots,65536\}, \{65537, 65538, \ldots, 2^{65536}\}, \ldots$$

Now, if a node has lies in a bucket $\{k+1, \ldots, 2^k\}$, it is credit 2^k units for future Find operations. Since we know that the number of nodes with rank k is at most $n/2^k$, the total amount that has been credited will be at most

$$2^k \sum_{i \geq k+1} \frac{n}{2^i} \leq 2^k \cdot \frac{n}{2^k} = n.$$

Now, there are at most $\log^* n$ buckets - this follows from the construction of the buckets and the definition of the iterated logarithm function. Thus the total amount credited across the entire sequence of operations is at most $n \log^* n$.

For each Find operation, we will count the cost in two parts - one is the cost the operation pays and the other is the part that will be taken from the credited amount. Every edge that is traversed in the path to the root that connects nodes with ranks in two different buckets will be paid by the Find operation. Since there are at most $\log^* n$ such edges, this cost that is paid is at most $\log^* n$. For every other edge from a node u to u.parent, the cost is paid by u from the amount which was credited to it.

But why is the credit given to u sufficient for all such traversals? To see this, let us assume that rank(u) lies in the bucket $\{k+1, \ldots, 2^k\}$. Since the rank of u.parent increases after each traversal of the edge between u and

u.parent, after at most 2^k such traversals, the rank(u) and rank$(u.\text{parent})$ are in different buckets. Since u has been credited 2^k units, this would be sufficient for all these traversals. Post this, the cost to traverse the edge between u and u.parent will be accounted in the Find operation itself. Thus we have the following statement about the amortized complexity of the path compression heuristic.

THEOREM 5.6. *The amortized complexity of Find operation when using union-by-rank with path compression on sets containing n elements is $O(\log^* n)$.*

Kruskal's algorithm

For a weighted graph $G(V, E, w)$ where $w : E \to \mathbb{R}$, and a spanning tree T of G, the weight of the spanning tree $w(T)$ is defined as $\sum_{e \in T} w(e)$ (the weight of the edges in T). The minimum spanning tree (MST) problem is to find a spanning tree of G with the smallest weight. We will assume that the weights are distinct, and hence we want to find the unique MST.

Recall the greedy-choice property of MSTs that form the basis for all the algorithms.

> Let $S \subseteq V$ be any subset of the vertex set, and let $\overline{S} = V - S$. The minimum weight edge $e \in E(S, V - S)$ is part of the MST.

Kruskal's algorithm for MST first sorts the edges according to edge weights. It then iterates over the list, adding an edge into the MST if the end points of the edge do not lie in the same connected component. The correctness of the algorithm follows from the greedy-choice property since each edge that is added is the minimum weight edge crossing the corresponding cut and must be in the MST.

To implement this algorithm, we start by sorting the edges according to their weights. This incurs a time complexity of $O(|E| \log |E|)$. Subsequently we can maintain a collection of sets corresponding to the connected components in the spanning forest as the edges are added to it. Initially the collection consists of the singleton sets $\{v\}$ for each $v \in V$. Now for each edge $\{u, v\} \in E$, we perform Find(u) and Find(v) to check if there are in the same component. If not, the edge $\{u, v\}$ is added to the MST and the Union(u, v) is performed. Using the tree representation of disjoint sets together with union-by-rank with path compression, the MST can be computed in time $O(|E| \log^* |V|)$. Thus, the total running time of Kruskal's algorithm is $O(|E| \log |E| + |E| \log^* |V|)$.