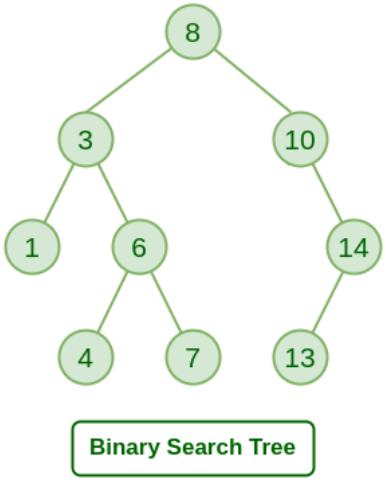
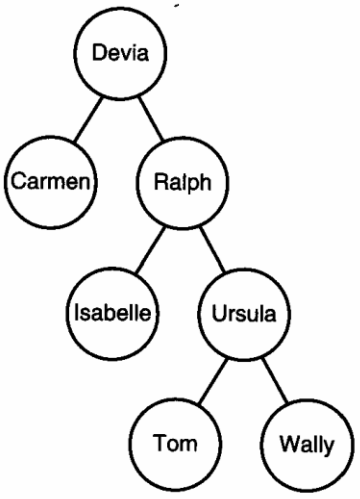
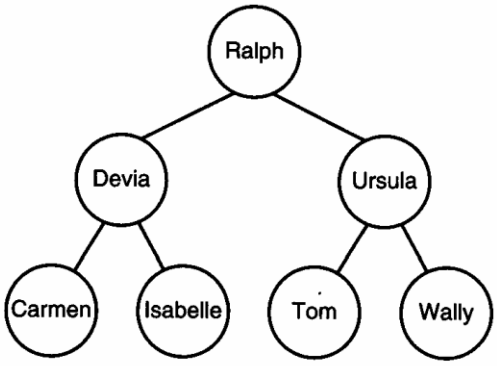
درخت های جستجوی دودویی

درخت جستجوی دودویی یک ساختار داده ای است که برای سازماندهی و ذخیره داده ها به شیوه ای مرتب شده استفاده می شود.

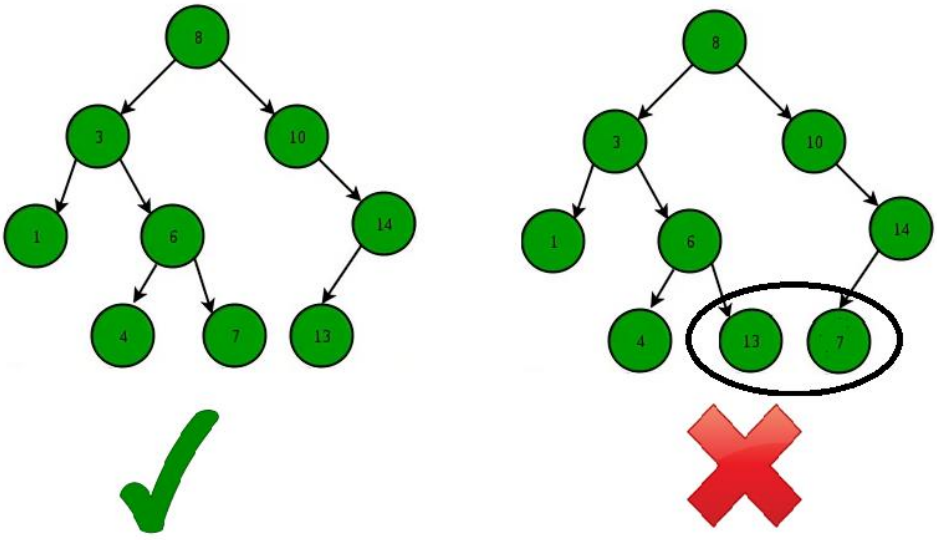
هر گره در درخت جستجوی باینری حداکثر دو فرزند دارد، یک فرزند چپ و یک فرزند راست، که فرزند چپ حاوی مقادیر کمتر از گره والد و فرزند راست حاوی مقادیر بزرگتر از گره والد است .این ساختار سلسله مراتبی امکان جستجوی کارآمد، درج و حذف عملیات روی داده های ذخیره شده در درخت را فراهم می کند.







زیردرخت چپ و راست هر کدام باید یک درخت جستجوی باینری باشند .هیچ گره تکراری نباید وجود داشته باشد:



- عمق (سطح) یک گره: تعداد لبه های موجود در مسیر منحصر بفرد از ریشه به آن گره.
- عمق یک درخت: حداکثر عمق تمامی گره ها در آن درخت.
- درخت دودویی متوازن: اگر عمق دو زیر درخت از هر گره بیش از یک واحد اختلاف نداشته باشند.
- درخت جستجوی دودویی بهینه: زمان متوسط برای مکان یابی یک کلید کمینه است.

Tabular difference between array and tree:

Parameter	Array	Tree
Nature	It is a linear data structure	It is a linear non-linear data structure
Base Notion	0th Index of the array	The root of the Tree
Successor	Element at reference_index + 1	Child nodes of the current node.
Predecessor	Element at reference_index – 1	Parent of the current node.
Natural Intuition	Staircase model with the base staircase as the i_{th} index	The best example to visualize the tree data structure is to visualize a natural rooted tree.
Order of Insertion	Usually an element inserted at current_index + 1	Depends on the type of tree.

Order of Deletion	At any index but after deletion elements are rearranged	Depends on the type of tree.
Insertion Complexity	O(1)->Insertion at end. O(N)->Insertion at random index.	Depends on the type for example AVL- $O(\log_2 N)$.
Deletion Complexity	O(1)->Deletion from end. O(N)->Deletion from a random index.	Depends on the type for example AVL- $O(\log_2 N)$.
Searching	O(N)	Depends on the type for example AVL- $O(\log_2 N)$.
Finding Min	O(N)	Depends on the type for example Min Heap- $O(\log_2 N)$.
Finding Max	O(N)	Depends on the type for example Max Heap- $O(\log_2 N)$.
IsEmpty	O(1)	Mostly O(1)
Random Access	O(1)	Mostly O(N)
Application	Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues and stacks.,	Fast search, insert, delete, etc.

یک BST عملیاتی مانند جستجو، درج، حذف، کف، سقف، بزرگتر، کوچکتر و غیره را در زمان O(h) که h ارتفاع BST است پشتیبانی می کند.

Examples of Binary Trees in real life include search systems like **online dictionaries**, decision trees in decision analysis, and sorting algorithms like heapsort. In Computer Science, Binary Trees are used in file system hierarchies, expression parsing in compiler design, and in network routers for data routing.

متوسط زمان جستجو

- زمان جستجو: تعداد مقایسه های انجام شده برای مکان یابی یک کلید
- زمان جستجو برای key برابر است با:

$$depth(key) + 1$$

- متوسط زمان جستجو:

$$\sum_{i=1}^{n} c_i \, p_i$$

که در آن:

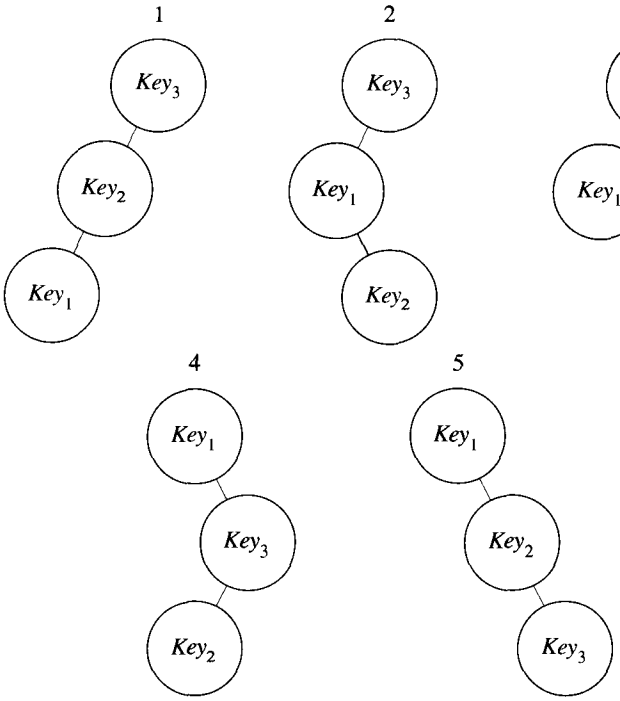
n تعداد کلید ها،

p_i احتمال آنکه key_i کلید مورد جستجو باشد،

c_i تعداد مقایسه های مورد نیاز برای یافتن key_i می باشد.

مثال:

$key_1 < key_2 < key_3$



$p_1 = 0.7$ $p_2 = 0.2,$ and $p_3 = 0.1,$

متوسط زمان جستجو برای 5 درخت:

1. $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2. $2(0.7) + 3(0.2) + 1(0.1) = 2.1$
3. $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
4. $1(0.7) + 3(0.2) + 2(0.1) = 1.5$
5. $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

هدف ساختن یک درخت جستجوی دودویی بهینه است که میانگین یافتن یک کلید درآن حداقل باشد.

به روش brute force:

Let $f(n)$ denote the total number of unique BST structures for number n. For a list of n unique numbers, we can choose any number as the root. For example, we can choose the number i ($i \in [1, n]$) as the root. Then, numbers in $[1, i-1]$ will be in the left subtree. Also, numbers in $[i+1, n]$ will be in the right subtree.

Since we have $(i-1)$ numbers in the left subtree, the left subtree has $f(i-1)$ unique BST structures. Similarly, the right subtree has $f(n-i)$ unique tree structures. Also, the arrangements in the left and right subtrees are independent. Therefore, we have a total of $f(i-1) \times f(n-i)$ unique tree structures when i is the root.

Since we have n possible ways to choose the root, we can add together $f(i-1) \times f(n-i)$ for all possible values of i:

$$f(n) = \sum_{i=1}^{n} f(i-1) f(n-i)$$

If we don't have any nodes in one subtree, then we'll only consider the total number of unique tree structures in the other subtree. Therefore, the base case of the formula is $f(0) = 1$.

تعداد حالات $C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{2n!}{n!(n+1)!}$

$C_n = \sum_{i=1}^{n} C_{i-1} C_{n-i}$

رویکرد برنامه ریزی پویا

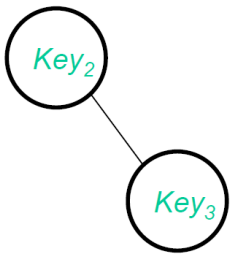
فرض کنید Key_i تا Key_j در درخت مرتب شده اند بطوریکه مقدار $\sum_{m=i}^{j} c_m p_m$ مینیمم شود که در آن c_m تعداد مقایسه برای یافتن Key_m در درخت است. یک چنین درختی را درخت بهینه برای آن کلیدها گوییم و مقدار بهینه را با $A[i][j]$ نشان می دهیم. همچنین

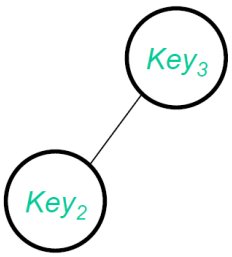
$$A[i][i] = p_i.$$

مثال:

- محاسبه $A[2][3]$ در مثال قبل

- $p_1 = 0.7, \quad p_2 = 0.2, \quad p_3 = 0.1$





$1(p_2) + 2(p_3) = 1(0.2) + 2(0.1) = 0.4$ $\qquad$ $2(p_2) + 1(p_3) = 2(0.2) + 1(0.1) = 0.5$

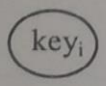
درخت سمت چپ بهینه است و بنابراین:

$$A[2][3] = 0.4$$

چون اصل بهینگی در اینجا صادق است می‌توانیم یک الگوریتم بازگشتی برای حل مسأله بنا می‌کنیم.
مشابه مسأله ضرب ماتریس‌ها از یک ماتریس A استفاده می‌کنیم که مفهوم عناصر آن به صورت زیر است:

A[i] [j] =	مینیمم میانگین زمان جستجو برای یک درخت BST
$1 \le i \le j \le n$, $key_i \le key_j$	با کلیدهای key_i تا key_j که

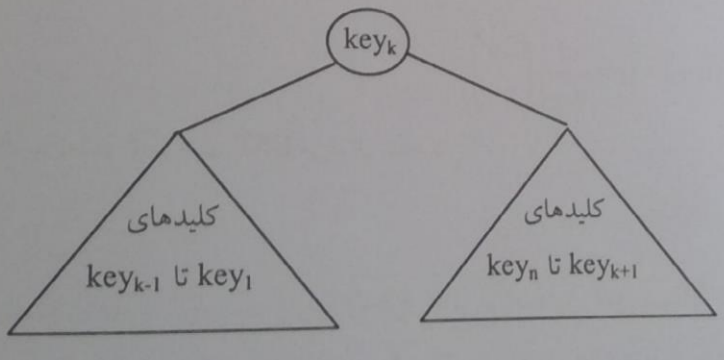
بدیهی است که $A[i][i] = P_i$ است چرا که در این حالت درخت BST فقط یک کلید key_i به صورت روبه‌رو دارد :



که میانگین زمان جستجوی آن برابر است با :

$$A[i][i] = 1 \times P_i = P_i$$

بنابر اصل بهینگی اگر درخت بهینه مورد نظر شامل ریشه key_k باشد انگاه دو زیر درخت چپ و راست این ریشه نیز باید BST بهینه باشند :



اگر درخت فوق، درخت بهینه BST باشد آنگاه A[1][n] با فرض آنکه key_k ریشه باشد برابر خواهد بود با:

$$\underbrace{A[1][k-1]}_{\text{زمان میانگین در زیر درخت چپ}} + \underbrace{P_1 + \ldots + P_{k-1}}_{\text{زمان اضافی مقایسه در ریشه}} + \underbrace{P_k}_{\text{زمان میانگین جستجو برای ریشه}} + \underbrace{A[k+1][n]}_{\text{زمان میانگین در زیر درخت راست}} + \underbrace{P_{k+1} + \ldots P_n}_{\text{زمان اضافی مقایسه در ریشه}}$$

$$= A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} P_m$$

توجه کنید فرمول فوق برای وقتی است که فرض کرده‌ایم key_k ریشه باشد ولی هر یک از key ها (key_1 تا key_n) می‌توانند در ریشه باشند که آنگاه جواب نهایی مینیمم آنها خواهد بود، لذا :

$$A[1][n] = \min_{1 \le k \le n} (A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} P_m$$

در حالت کلی برای هر key_i تا key_j داریم :

$$A[i][j] = \min_{i \le k \le j} (A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} P_m \qquad (i < j)$$

$$A[i][i] = P_i \qquad (i = j)$$

الگوریتم

▶ Algorithm 3.9 **Optimal Binary Search Tree**

Problem: Determine an optimal binary search tree for a set of keys, each with a given probability of being the search key.

Inputs: n, the number of keys, and an array of real numbers p indexed from 1 to n, where $p[i]$ is the probability of searching for the ith key.

Outputs: A variable *minavg*, whose value is the average search time for an optimal binary search tree; and a two-dimensional array R from which an optimal tree can be constructed. R has its rows indexed from 1 to $n+1$ and its columns indexed from 0 to n. $R[i][j]$ is the index of the key in the root of an optimal tree containing the ith through the jth keys.

```
void optsearchtree (int n,
                    const float p[],
                    float& minavg,
                    index R[][])
{
  index i, j, k, diagonal;
  float A[1..n + 1][0..n];

  for (i = 1; i <= n; i++){
     A[i][i - 1] = 0;
     A[i][i] = p[i];
     R[i][i] = i;
     R[i][i - 1] = 0;
  }
  A[n + 1][n] = 0;
  R[n + 1][n] = 0;
  for (diagonal = 1; diagonal <= n - 1; diagonal++)
     for (i = 1; i <= n - diagonal; i++){          // Diagonal-1 is
                                                   // just above the
                                                   // main diagonal.
        j = i + diagonal;
        A[i][j] = minimum(A[i][k - 1] + A[k + 1][j]) + sum_{m=i}^{j} p_m .
                  i <= k <= j
        R[i][j] = a value of k that gave the minimum;
     }
  minavg = A[1][n];
}
```

پیچیدگی زمانی:

- عمل اصلی: دستورالعمل های اجرا شده برای هر مقدار از k
- اندازه ورودی: n، تعداد کلید ها
- پیچیدگی زمانی:

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

ساختن درخت:

▶ Algorithm 3.10 **Build Optimal Binary Search Tree**

Problem: Build an optimal binary search tree.

Inputs: n, the number of keys, an array Key containing the n keys in order, and the array R produced by Algorithm 3.9. $R[i][j]$ is the index of the key in the root of an optimal tree containing the ith through the jth keys.

Outputs: a pointer *tree* to an optimal binary search tree containing the n keys.

```
node_pointer tree (index i, j)
{
  index k;
  node_pointer p;

  k = R[i][j];
  if (k == 0)
     return NULL;
  else{
     p = new nodetype;
     p-> key = Key[k];
     p-> left = tree(i, k - 1);
     p-> right = tree(k + 1, j);
     return p;
  }
}
```

مثال:

- کلید ها:

Don	Isabelle	Ralph	Wally
Key[1]	Key[2]	Key[3]	Key[4]
p_1=3/8	p_2=3/8	p_3=1/8	p_4=1/8

- آرایه های ایجاد شده:

$A_{1,4} = \min_{1 \le k < 4} (A_{1,k-1} + A_{k+1,4}) + \cdots$

	0	1	2	3	4
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

	0	1	2	3	4
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

R

درخت بهینه: