CS213/293 Data Structure and Algorithms 2025

Lecture 6: Binary search tree (BST)

Instructor: Ashutosh Gupta

IITB India

Compile date: 2025-09-13

Ordered dictionary

Recall: There are two kinds of dictionaries.

- Dictionaries with unordered keys
  - We use hash tables to store dictionaries for unordered keys.
- Dictionaries with ordered keys
  - Let us discuss the efficient implementations for them.

Recall: Dictionaries via ordered keys on arrays

- Searching is $O(log\ n)$
- Insertion and deletion is $O(n)$
  - Need to shift elements before insertion/after deletion

Can we do better?

Topic 6.1

Binary search trees

Binary search trees (BST)

Definition 6.1

A binary search tree is a binary tree T such that for each $n \in T$

- n is labeled with a key-value pair of some dictionary,
  - (if $label(n) = (k, v)$, we write $key(n) = k$)
- for each $n' \in descendants(\ left(n)\ )$, $key(n') \leq key(n)$, and
- for each $n' \in descendants(\ right(n)\ )$, $key(n') \geq key(n)$.

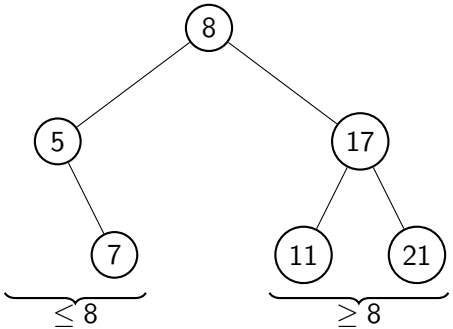
Note that we allow two entries to have the same keys. The same key can be in either of the subtrees.

Commentary: We assume $descendants(Null) = \emptyset$.

Example: BST

Example 6.1

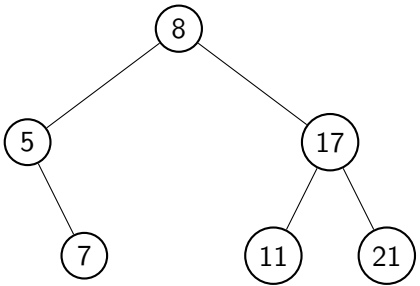
In the following BST, we show only keys stored at the node.

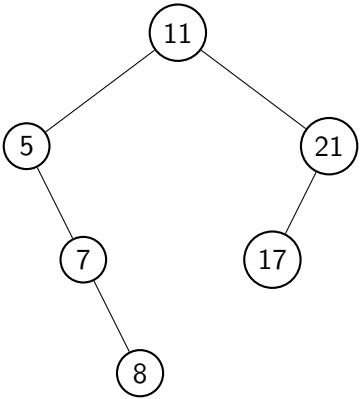


Example: many BSTs for the same data

Example 6.2

The same set of keys may result in different BSTs.

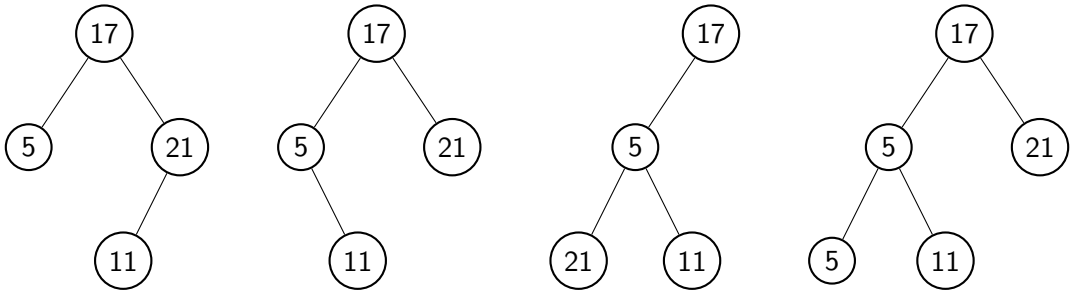




Exercise: Identify BST

Exercise 6.1

Which of the following are BSTs?



Topic 6.2

Algorithms for BST

Algorithms for BST

We need the following methods on BSTs

- search
- insert
- minimum/maximum
- successor/predecessor: Find the successor/predecessor key stored in the dictionary
- delete

Exercise 6.2

Give minimum and successor algorithms for sorted array-based implementation of a dictionary.

Commentary: Recall that we did not discuss algorithms for minimum and successor in our earlier discussion of unordered dictionaries, which are implemented using hash tables. Since we cannot define successor and minimum for unordered keys, the question of such algorithms does not arise. However, we do need them for operations on BSTs.

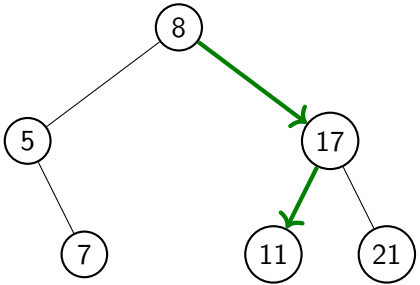
Searching in BST

Commentary: By the definition of BST, we are guaranteed that 11 will not occur in the left subtree of 8. This is the same reasoning as the binary search that we discussed earlier.

Example 6.3

Searching 11 in the following BST.

- We start at the root, which is node 8
- At node 8, go to the right child because $11 > 8$.
- At node 17, go to the left child because $11 < 17$.
- We find 11 at the node.

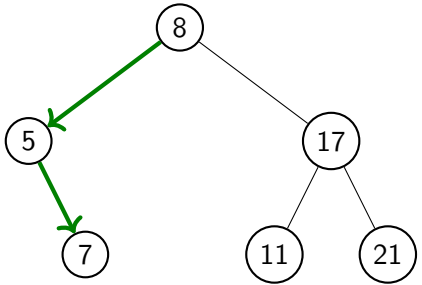


Unsuccessful search in BST

Example 6.4

Searching 6 in the following BST.

- We start at the root, which is node 8
- At node 8, go to the left child because $6 < 8$.
- At node 5, go to the right child because $6 > 5$.
- At node 7, go to the left child because $6 < 7$.
- Since node 7 has no left child the search fails.



Algorithm: Search in BST

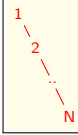
Algorithm 6.1: SEARCH(BST T, int k)

```
1 n := root(T);
2 while n ≠ Null do
3     if key(n) = k then
4         break
5     if key(n) > k then
6         n := left(n)
7     else
8         n := right(n)
9 return n
```

- Running time is $O(h)$, where h is height of BST.
- If there are n keys in the BST, the worst case running time is $O(n)$.

Commentary: Answer:
a. We search in the BST. If the key is found on a node, then we start two(Why?) searches in both the subtrees of the found node. We recursively start the searches.

b. Find N in the following BST



Exercise 6.3

a. Modify the above algorithm to find all occurrences of key k.
b. Give an input of SEARCH that exhibits worst-case running time.

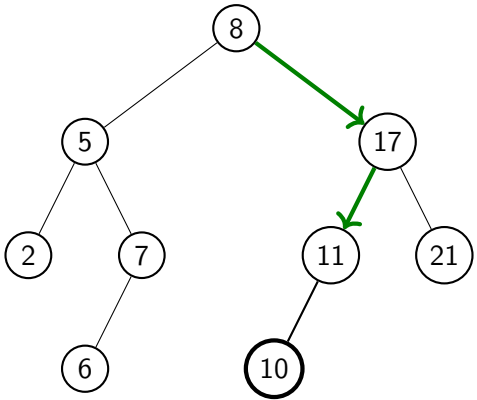
Topic 6.3

Insert in BST

Example: Insert in BST

Example 6.5

key 10 was not there in the BST and we want to insert 10. Where do we insert 10?



We always insert the new key as leaf.

Algorithm: Insert in BST

Algorithm 6.2: INSERT(BST T, Node n)

```
1  x := root(T); y := Null;
2  while x ≠ Null do
3  |   y := x;
4  |   if key(x) > key(n) then
5  |   |   x := left(x)
6  |   else
7  |   |   x := right(x)
8  if y = Null then
9  |   root(T) = n;
10 if key(y) > key(n) then
11 |   left(y) := n
12 else
13 |   right(y) := n
14 parent(n) = y
```

Exercise 6.4

a. What is the running time of the algorithm?
b. Give an order of insertion for the maximum tree height.
c. Give an order of insertion for the minimum tree height.
d. What does happen if key(n) already exists?

Commentary: Answer:
a. the same as search,
b. 1,2,3,4,5,...,n
c. n/2,n/4,3n/4,n/8,3n/8,5n/8,7n/8,..
d. This algorithm always goes right. It is correct but may not be a good idea. It should randomly choose left or right.

Topic 6.4

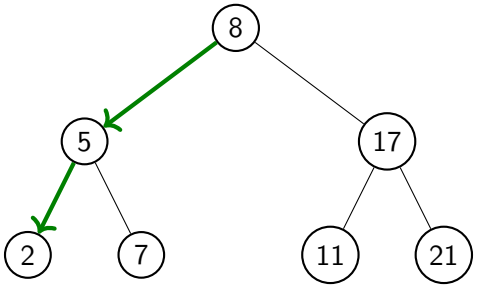
Minimum in BST

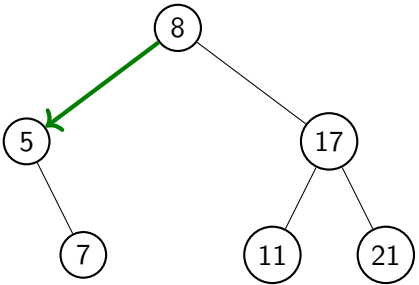
Example: minimum in BST

Commentary: Always go left to find a smaller node. As soon as we do not have a left child, we have found the minimum node.

Example 6.6

What is the minimum of the following BSTs?





Algorithm: Minimum in BST

The following algorithm computes the minimum in the subtree rooted at node n.

Algorithm 6.3: MINIMUM(Node n)

```
1 while n ≠ Null and left(n) ≠ Null do
2     n := left(n)
3 return n
```

- Runtime analysis is the same as SEARCH.

Exercise 6.5

Modify the above algorithm to compute the maximum

Correctness of MINIMUM

Commentary: Note that $key(n') \leq key(n) \leq key(n'')$ where $n'' \in descendants(right(n))$

Theorem 6.1

If $n \neq Null$, the returned node by MINIMUM(n) has the minimum key in the subtree rooted at n.

Proof.

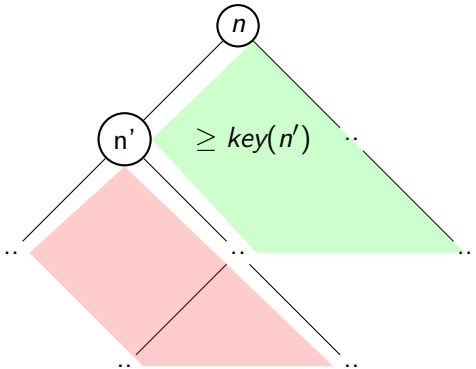
If $left(n) = Null$, $key(n)$ is the minimum key.

Otherwise, we go to $n' = left(n)$. Any node not in $descendants(n')$ must have a larger key than $key(n')$.(Why?)

So the minimum of $descendants(n')$ is the overall minimum.

This argument continues to hold for any number of iterations of the loop. (induction)

Therefore, our algorithm will compute the minimum.



Topic 6.5

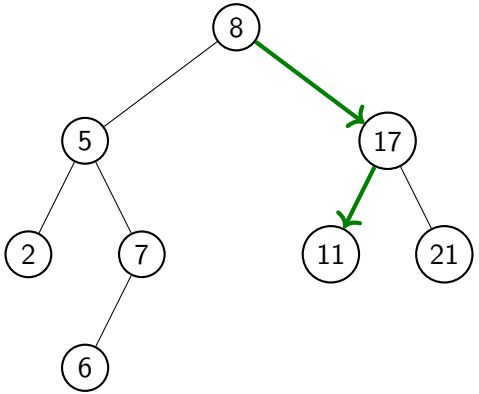
Successor in BST

Example: successor in BST

We now consider the problem of finding the node that has the successor key of a given node.

Example 6.7

Where is the successor of 8?

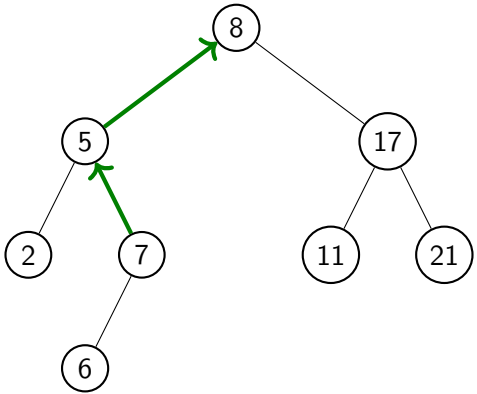


Observation: Minimum of right subtree.

Example: successor in BST(2)

Example 6.8

Where is the successor of 7?



Exercise 6.6

a. When do we not have the successor in the right subtree?

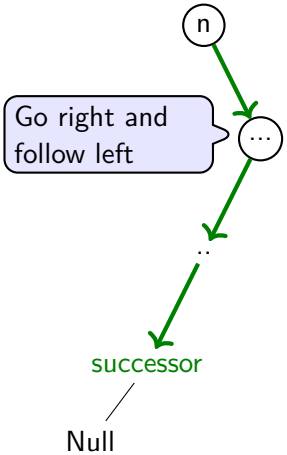
b. If the successor is not in the right subtree, where else can it be?

Cases for the location of the successor

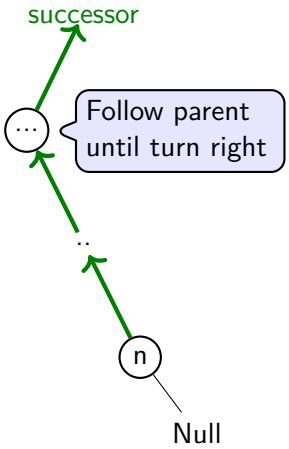
Finding a successor to n

Commentary: Successor may be found in two possible areas. The second case is slightly difficult to understand, where the successor is one of the ancestors. It is the closest ancestor that is bigger than n. This happens when the path turns right first time. The formal proof is at the end of the slides.

Case 1: If there is a right subtree:



Case 2: If there is no right subtree:



Successor in BST

Algorithm 6.4: SUCCESSOR(BST T, node n)

if $right(n) \neq Null$ **then**
  return MINIMUM($right(n)$)
while $parent(n) \neq Null$ *and* $right(parent(n)) = n$ **do**
  $n := parent(n)$;
return $parent(n)$

Exercise 6.7

a. Modify the above algorithm to compute predecessor
b. What is the running time complexity of SUCCESSOR?
c. What happens when we do not have any successor?
d. What is returned if multiple keys have the same value?
e. What is the connection between the above algorithm and in-order walk?
f. Can we modify the above algorithm to find the strict successor?

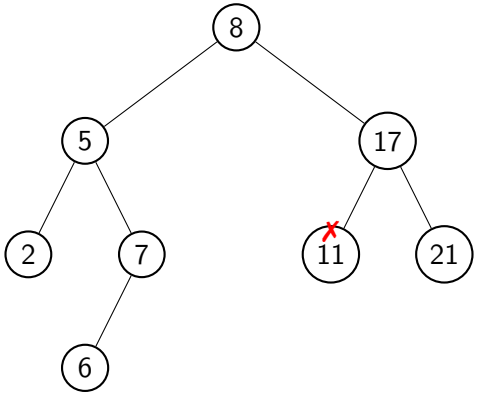
Topic 6.6

Deletion

Example: deleting a leaf

Example 6.9

How can we delete leaf 11?

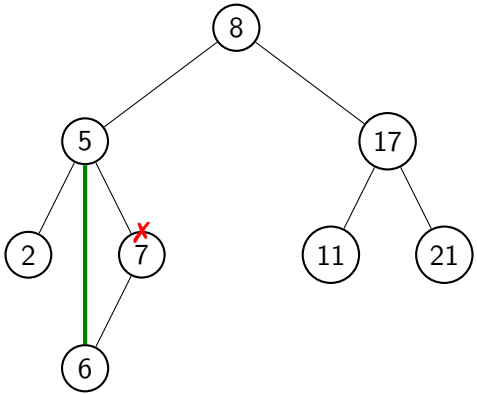


We delete leaf 11 by simply removing the node.

Example: deleting a node with a single child

Example 6.10

How can we delete node 7, which has a single child?

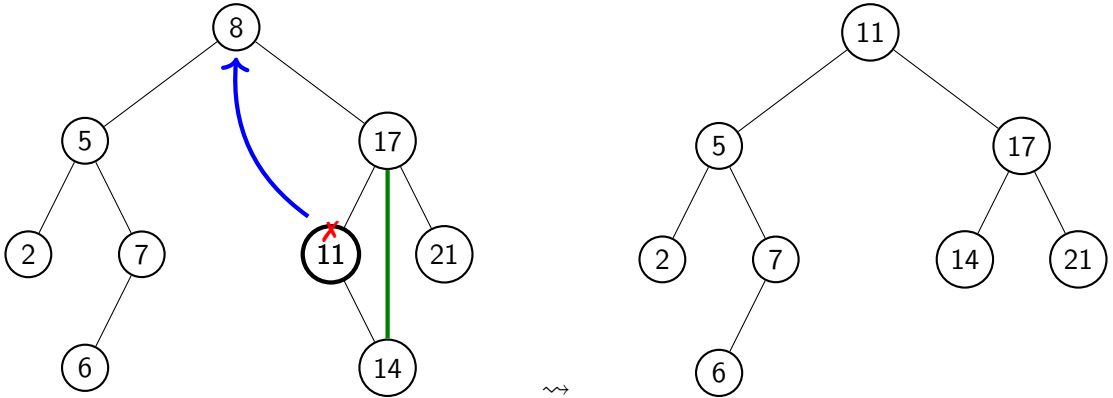


We delete node 7 by making 6 the child of 5 and removing the node.

Example: deleting a node with both children

Example 6.11

How can we delete node 8, which has both the children?



We delete node 8 by removing 11, which is the successor of 8, and moving the data of 11 to 8.

Algorithm: delete in BST*

Algorithm 6.5: DELETE(BST T, Node n)

$y := (left(n) = Null \vee right(n) = Null)\ ?\ n\ :\ \text{SUCCESSOR}(T, n)$; // y will be deleted

if $y \neq n$ **then**

  $key(n) := key(y)$ // copy all data on y

$x := (left(y) = Null)\ ?\ right(y)\ :\ left(y)$; //$x$ is the child of y or x is Null

if $x \neq Null$ **then**

  $parent(x) = parent(y)$ //y is not a leaf, update the parent of x

if $parent(y) = Null$ **then**

  $root(T) = x$ // y was the root, therefore x is root now

else

  if $left(parent(y)) = y$ **then**

    $left(parent(y)) := x$ //Remove y from the tree

  else

    $right(parent(y)) := x$ //Remove y from the tree

Exercise 6.8

a. How can we delete by key instead of node? Does it change the complexity? b. Do we need to free y?

Topic 6.7

Average BST depth**

How often do we get a small height?

All BST algorithms are $O(height)$.

Can we estimate the average cost of the operations?

Here is an analysis that suggests that most often BSTs have small height.

Exercise 6.9
Why are we saying that the analysis suggests instead of proves?

Average cost of n-inserts

Let us consider a random permutation of $1, .., n$ and an empty BST.

We insert the sequence of numbers in the BST.

The total cost of insertions will be the sum of the levels of nodes in the resulting BST.

Definition 6.2

Let $T(n)$ denote the average time taken over $n!$ permutations to insert n keys.

We will use the number of comparisons as the cost to insert a node.

Exercise 6.10

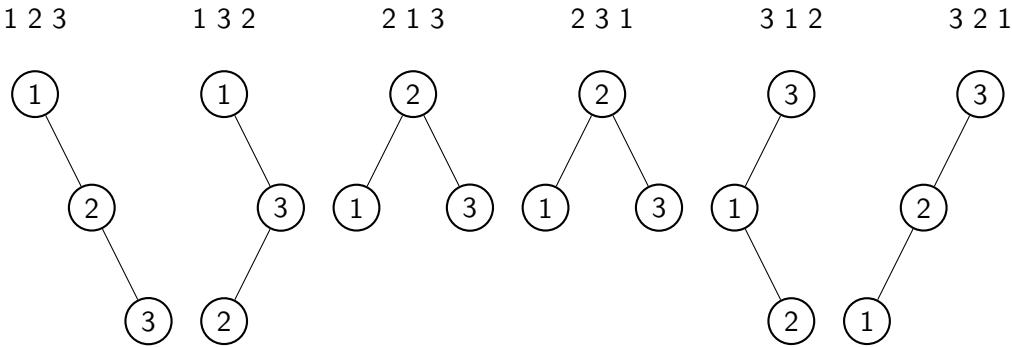
What are the best and worst insertion times?

Example: Computing $T(n)$

Commentary: The insertion cost of a key is the number of comparisons to insert the key. For example, in the first tree, we insert 1 without any comparison, 2 with one comparison, and 3 with two comparisons. In total, we have three comparisons.

Example 6.12

Let us compute the average of total cost of inserting three elements in an empty BST.



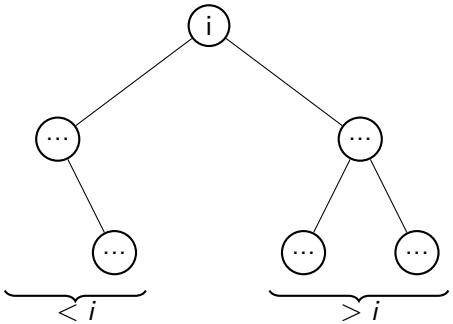
$T(3) = 16/6$

Recurrence for $T(n)$

Out of all $n!$ permutations, i is the first element in $(n-1)!$ permutations. (Why?)

In the BSTs generated for the permutations,

- i is the root,
- keys $1, ..., i-1$ are in the left subtree, and
- keys $i+1, ..., n$ are in the right subtree.

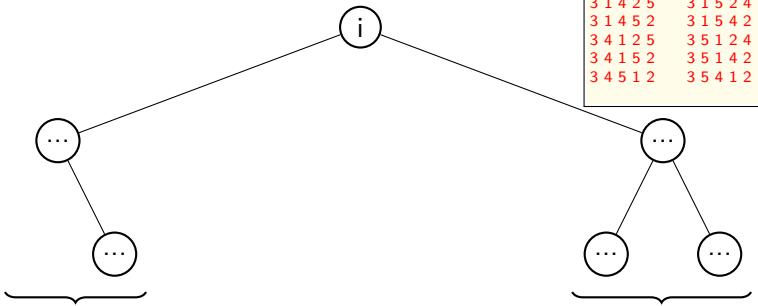


Recurrence for $T(n)(2)$

Note that there are $(i-1)!$ orderings for keys $1, ..., i-1$.

Commentary: The following counting argument of the number of embeddings of a permutation is non-trivial. Let $n = 5$ and $i = 3$. There are two permutations of 1 and 2, which are 1 2 and 2 1. There are $(5-1)! = 24$ permutations, where 3 occurs at the first position. Permutation 1 2 is embedded in the 12 permutations out of the 24, which are as follows.

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



In the $(n-1)!$ permutations, each permutation of $1, ..., i-1$ is embedded $\frac{(n-1)!}{(i-1)!}$ times.

In the $(n-1)!$ permutations, each permutation of $i+1, ..., n$ is embedded $\frac{(n-1)!}{(n-i)!}$ times.

Recurrence for $T(n)(3)$

While inserting keys $1, .., i-1$, each key is compared with root i, which is an additional unit cost per insertion.

Consider left subtree:

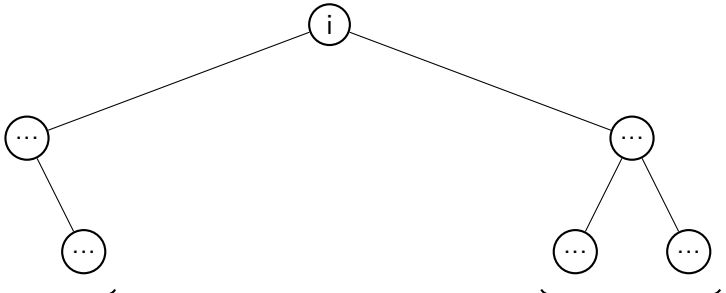
- In the subtree, the average of total insert time is $T(i-1)$.
- Since we have one more node above, the average of total insert time is $T(i-1)+i-1$.
- The total time to insert in all the orderings is $(i-1)!(T(i-1)+i-1)$.

Consider right subtree:

- In the subtree, the average insert time is $T(n-i)$.
- Since we have one more node above, the average of total insert time is $T(n-i)+n-i$.
- The total time to insert in all the orderings is $(n-i)!(T(n-i)+n-i)$.

Recurrence for $T(n)(4)$

Total time of insertion for all the $(n-1)!$ permutations, where i is the root.



$$\frac{(n-1)!}{(i-1)!} \times (i-1)!(T(i-1)+i-1) \quad + \quad \frac{(n-1)!}{(n-i)!} \times (n-i)!(T(n-i)+n-i)$$

$$= (n-1)!(T(i-1)+i-1) \quad + \quad (n-1)!(T(n-i)+n-i)$$

$$= (n-1)!(T(i-1)+T(n-i)+n-1).$$

Recurrence for $T(n)$(5)

The total time of insertions in all $n!$ permutations

$$\sum_{i=1}^{n} (n-1)!(T(i-1) + T(n-i) + n - 1).$$

Therefore, the average time of insertions in all permutations

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

After simplification,

$$T(n) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n - 1,$$

where $T(0) = 0$.

Commentary: $T(1) = 0$, $T(2) = 1$,
$T(3) = 2/3(1 + 0 + 0) + (3 - 1) = 2\frac{2}{3}$
$T(4) = 2/4(8/3 + 1 + 0 + 0) + (4 - 1) = 4\frac{10}{12}$

Exercise 6.11

Compute $T(3)$ via the above equation.

What is the growth of $T(n)$?

We need to find an approximate upper bound of $T(n)$.

Let us solve the recurrence relation.

Simplify the recurrence relation

Now we will derive the relation between $T(n)$ and $T(n-1)$.
Let us write the relation for $T(n-1)$.

$$T(n-1) = \frac{2}{n-1}\sum_{i=0}^{n-2} T(i) + n - 2.$$

After reordering the terms, we obtain

$$\sum_{i=0}^{n-2} T(i) = \frac{n-1}{2}(T(n-1) - n + 2).$$

Let us look at the expression for $T(n)$ again. After reordering of terms in $T(n)$,

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-2} T(i) + \frac{2}{n}T(n-1) + n - 1 = \frac{2}{n}\frac{n-1}{2}(T(n-1) - n + 2) + \frac{2}{n}T(n-1) + n - 1,$$

$$T(n) = \frac{n+1}{n}T(n-1) + \frac{n-1}{n}(-n+2) + n - 1 = \frac{n+1}{n}T(n-1) + \frac{2(n-1)}{n}.$$

Approximate recurrence relation

From

$$T(n) = \frac{n+1}{n} T(n-1) + \frac{2(n-1)}{n},$$

we can conclude

$$T(n) \leq \frac{n+1}{n} T(n-1) + 2.$$

Expanding the approximate recurrence relation

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

After a sequence of substitutions till $T(0)$, we obtain

$$
T(n) \leq \frac{n+1}{n-(n-1)} T(0) + \frac{n+1}{2} 2 + \ldots + \frac{n+1}{n} 2 + 2.
$$

Expanding the approximate recurrence relation

$$T(n) \leq 2(n+1)(\underbrace{\frac{1}{2} + \ldots + \frac{1}{n}}_{\leq ln\ n}) + 2$$

$$T(n) \leq 2(n+1)(ln\ n) + 2$$

Therefore,

$$T(n) \in O(n \log\ n)$$

Commentary: $\frac{1}{2} + \ldots + \frac{1}{n} \leq \int_1^n \frac{1}{x} dx = \ln n$

$T(1) \leq 2$

$T(2) \leq 2 * (2+1)(1/2) + 2 = 3 + 2 = 5$

$T(3) \leq 2*(3+1)(1/2+1/3)+2 = 8*5/6+2 = 8\frac{4}{6}$

$T(4) \leq 2 * (4+1)(1/2+1/3+1/4) + 2 = 12\frac{10}{12}$

Run the following Python code to numerically visualize the gap in the approximation.

```
Texact  = [ 0.0, 0.0]
Tapprox = [ 0.0, 2  ]
fsum = 0.0
for n in range(2,1000):
  Texact.append((n+1)/n*Texact[n-1]+2*(n-1)/n)
  fsum += 1/n
  Tapprox.append(2*(n+1)*fsum+2)
  print(Tapprox[n]/Texact[n])
```

Exercise 6.12

Prove/Disprove $T(n) \in \Theta(n \log n)$. [Hint: Consider $T(n) \geq \frac{n+1}{n} T(n-1) + \alpha$, where $\alpha < 2$ and $n > 2/(2-\alpha)$]

Topic 6.8

Tutorial problems

Exercise: sorting via BST

Exercise 6.13

a. Show that in order printing of BST nodes produces a sorted sequence of keys.
b. Give a sorting procedure using BST.
c. Give the complexity of the procedure.

Exercise: delete all smaller keys

Exercise 6.14

Given a BST T and a key k, the task is to delete all keys less than k from T. Please write a pseudocode for this. What is the running time of your algorithm? What is the structure of the leftover tree? What is the root of the tree?

Exercise: expected height

Exercise 6.15

Let H(n) be the expected height of the tree obtained by inserting a random permutation of 1,...,n. Write the recurrence relation for H(n).

Exercise: find the leftmost and rightmost

Exercise 6.16

Given a BST tree T and a value v, write a program to locate the leftmost and rightmost occurrence of the value v.

Topic 6.9

Problems

True or False

Exercise 6.17

Mark the following statements True / False and also provide justification.

1. Binary search trees are always balanced.
2. In binary search tree containing n nodes, the worst-case running time of insertion is $O(\log n)$.

Exercise: post-order search tree

Exercise 6.18

Consider a binary tree with labels such that the postorder traversal of the tree lists the elements in increasing order. Let us call such a tree a post-order search tree. Give algorithms for search, min, max, insert, and delete on this tree.

Exercise: permutations

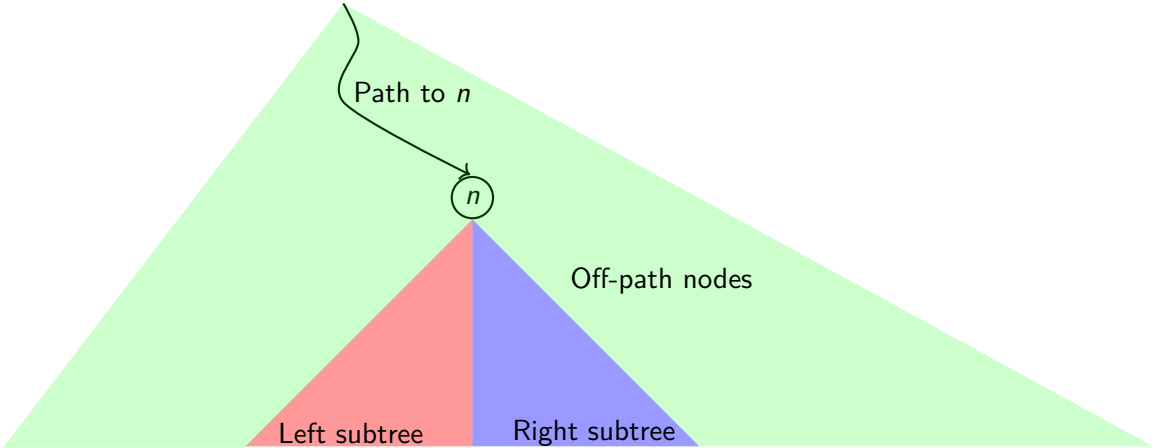
Exercise 6.19

Let [a(1),...,a(n)] be a random permutation of n. Let p(i) be the probability of event a(a(1))=i. Compute p(i).

Topic 6.10

Extra slides: proof of correctness of successor

Parts of BST with respect to a node n



The least common ancestor(LCA) is in the middle

Theorem 6.2

For nodes n_1 and n_2, let $n = LCA(n_1, n_2)$. If $key(n_1) \leq key(n_2)$, $key(n_1) \leq key(n) \leq key(n_2)$.

Proof.

We have four cases.

case $n_1 \in ancestors(n_2)$: Trivial.(Why?)

case $n_2 \in ancestors(n_1)$: Trivial.

case $key(n_1) = key(n_2)$:
Since $key(n)$ divided one of the nodes to left and the other to right, $key(n) = key(n_1)$.

case $key(n_1) < key(n_2)$:
n_1 and n_2 must be in the left and right subtree of n respectively.
Therefore, $key(n_1) \leq key(n) \leq key(n_2)$. □

Larger ancestors keep growing!

Theorem 6.3
If $n_2 \in ancestors(n_1)$, $n_1 \in ancestors(n)$, and $key(n_2) > key(n)$, then $key(n_2) \geq key(n_1)$.

Proof.
n must be in the left subtree of n_2.

n_1 must be in the subtree.(Why?)

Since n_1 is in the left subtree of n_2, $key(n_2) \geq key(n_1)$. □

Correctness of SUCCESSOR

In the following proof, we assume that all nodes have distinct elements.

Theorem 6.4

Let T be a BST, node $n \in T$, and $n' = \text{SUCCESSOR}(n)$.
If $n' \neq Null$, $key(n') > key(n)$ and for each node $n'' \in T - \{n, n'\}$, we have

$$\neg(key(n) < key(n'') < key(n')).$$

Proof.

Claim: A successor of n cannot be an off-path node.
Assume an off-path node n' is the successor of n.
Therefore, $key(n) < key(n')$.
Due to theorem 6.2, $key(n) \leq key(LCA(n, n')) \leq key(n')$.
Therefore, $key(LCA(n, n'))$ is between the nodes. Contradiction. ...

Correctness of SUCCESSOR(2)

Proof(Continued).

Claim: Successor of n cannot be in left subtree.
All nodes will have keys that are less than $key(n)$.

Claim: If the right subtree exists, then a successor cannot be on the path to n.

1. Consider $n' \in descendants(right(n))$.
2. Therefore, $key(n') > key(n)$.
3. For some $n'' \in ancestors(n)$, let us assume n'' is successor of n.
4. Therefore, $key(n'') > key(n)$.
5. Therefore, $n \in descendants(left(n''))$.
6. Therefore, $n' \in descendants(left(n''))$. due to 1 and 5
7. Therefore, $key(n'') > key(n')$.
8. Therefore, $key(n'') > key(n') > key(n)$.
9. Therefore, $key(n'')$ is not a successor. Contradiction.

Correctness of SUCCESSOR(2)

Proof(Continued).

Claim: If the right subtree exists, the successor is the minimum of the right subtree.
Since the successor is nowhere else, it must be the minimum.

Claim: If there is no right subtree and there is a node greater than n, the successor is the closest node on the path to n such that the key of the node is greater than n.
Let $n_1, n_2 \in ancestors(n)$ such that $n_2 \in ancestors(n_1)$, $key(n_2) > key(n)$, and $key(n_1) > key(n)$.
Due to theorem 6.3, $key(n_2) > key(n_1)$.
Therefore, n_2 cannot be a successor.
Therefore, the closest node to n is the successor. □

Exercise 6.20

a. Show that the closest node in the above proof must have n in its right subtree.
b. There is a final case missing in the above proof. What is the case? Prove the case.
b. Modify the above proof to support repeated elements in BST.

End of Lecture 6