CS213/293 Data Structure and Algorithms 2024

Lecture 6: Binary search tree (BST)

Instructor: Ashutosh Gupta

IITB India

Compile date: 2024-08-20

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* $n' \in descendants(\ right(n)\ )$*,* $key(n') \geq key(n)$*.*

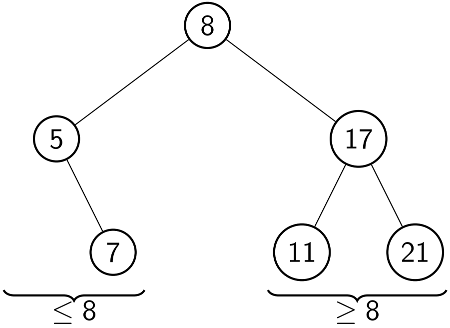
Note that we allow two entries to have the same keys. The same key can be in either of the subtrees. Just maintain it consistently.

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

Example: BST

Example 6.1

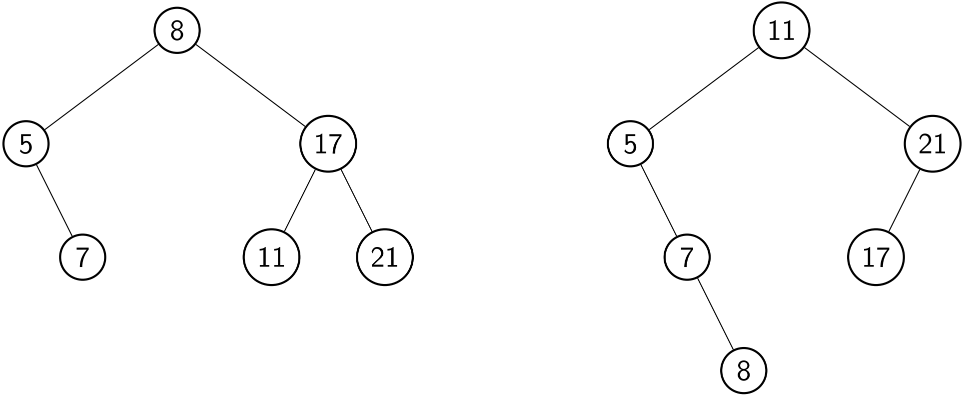
In the following BST, we are showing only keys stored at the node.



Example: many BSTs for the same data

Example 6.2

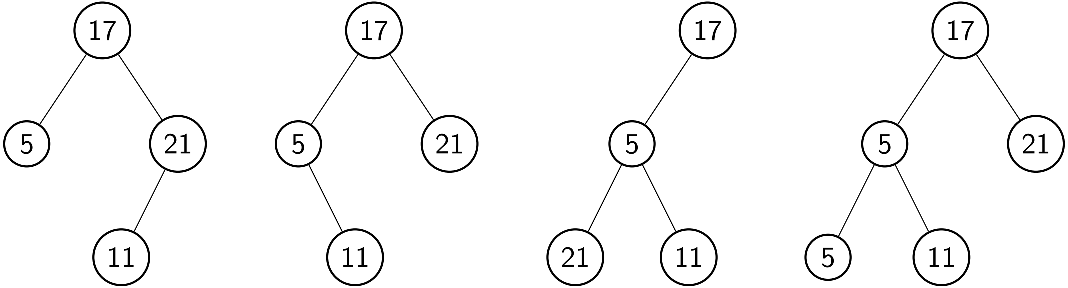
The same set of keys may result in different BSTs.



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

Exercise 6.2

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

It is sorted means 0th index is minimum, and successor is required index+1

Commentary: We did not discuss algorithms for minimum and successor in our earlier discussion of unordered dictionaries. However, we need them for other operations on BST.

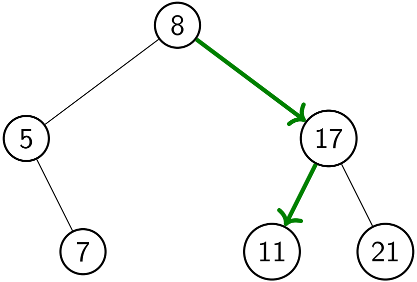
Searching in BST

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.

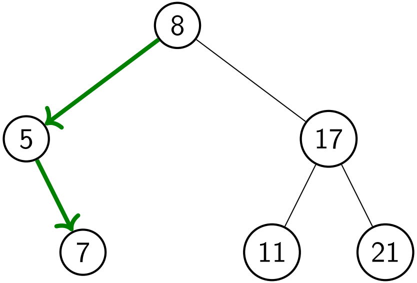


Unsuccessful search in BST

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)

```
n := root(T);
while n ≠ Null do
    if key(n) = k then
        break
    if key(n) > k then
        n := left(n)
    else
        n := right(n)
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

```
1
 \
  2
   \
    ..
     \
      N
```

Exercise 6.3

a. Modify the above algorithm to find all occurrences of key k.

Recursion would handle it!

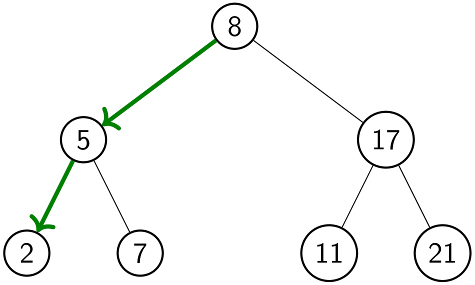
b. Give an input of SEARCH *that exhibits worst-case running time.*

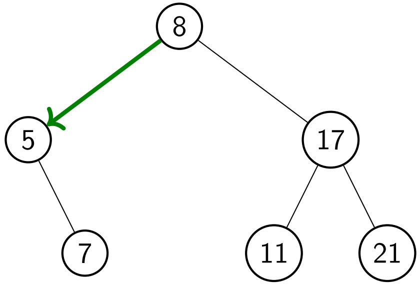
Example: minimum in BST

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 6.5

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 6.2: MINIMUM(Node n)

while $n \neq Null$ *and* $left(n) \neq Null$ **do**
  $n := left(n)$
return n

- Runtime analysis is same as SEARCH.

Exercise 6.4

Modify the above algorithm to compute the maximum Go right

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

Correctness of MINIMUM

Theorem 6.1

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

Proof.

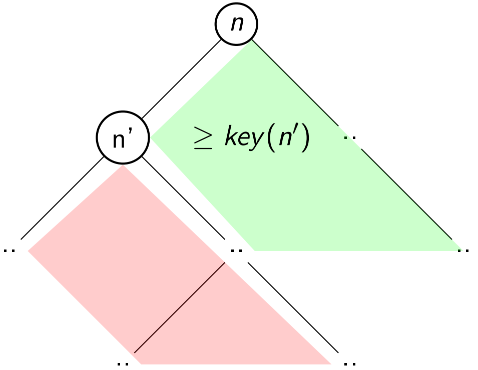
If $left(n) = Null$, $key(n)$ is the minimum key.

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

So 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.



□

Successor in BST

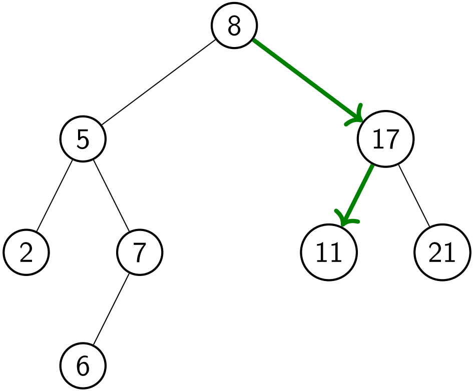
When we're talking about successor of a node, we mean the node just later the given node in Inorder traversal!

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

Example: successor in BST

Example 6.6

Where is the successor of 8?

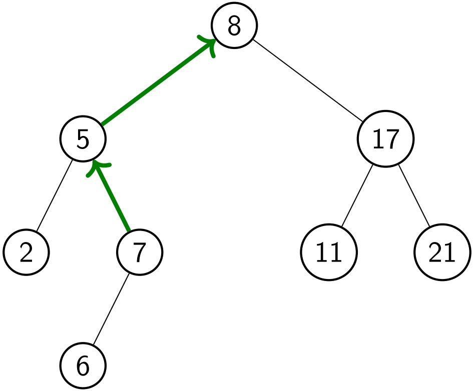


Observation: Minimum of right subtree.

Example: successor in BST(2)

Example 6.7

Where is the successor of 7?



Exercise 6.5

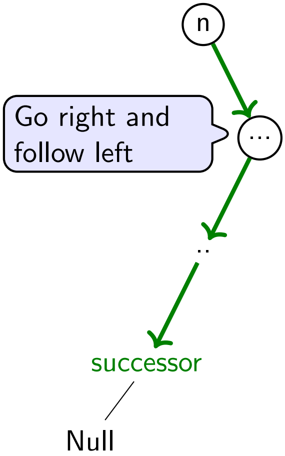
a. *When do we not have the successor in the right subtree?* When it doesn't exist

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

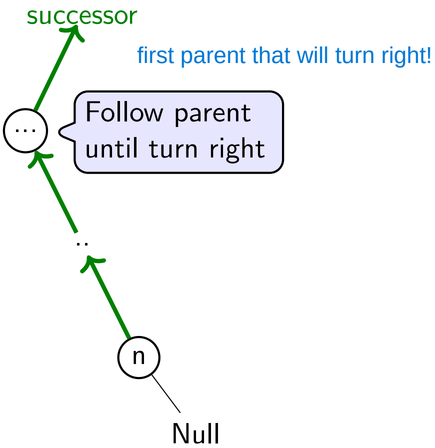
Successor locations

Finding successor of *n*

Case 1: If there is a right subtree:



Case 2: If there is no right subtree:



Successor in BST

Algorithm 6.3: 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 $parent(n)$

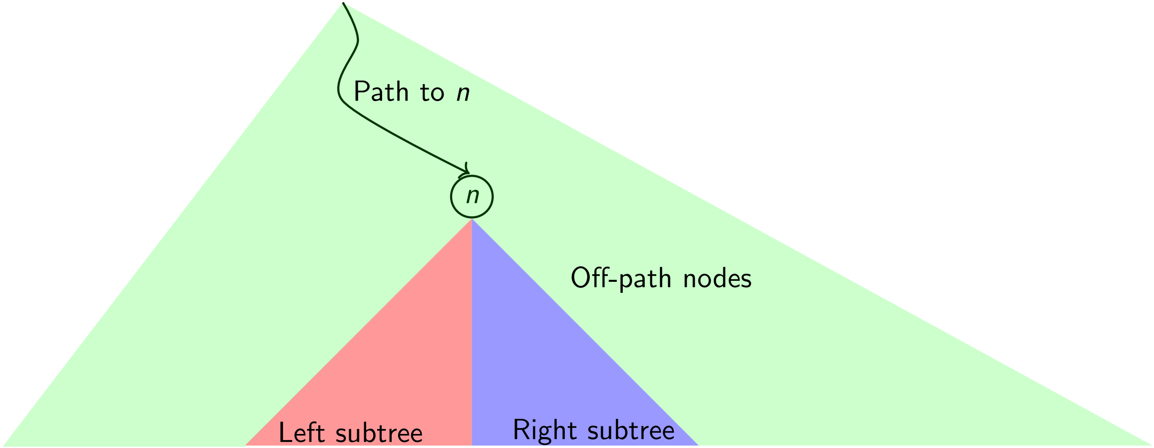
Exercise 6.6

a. *Modify the above algorithm to compute predecessor* just replace the right->left and min->max

b. *What is the running time complexity of* SUCCESSOR*?* kind of log(n)

c. *What happens when we do not have any successor?* it returns null

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

$n_1 \in ancestors(n)$ and $n_2 \in ancestors(n_1)$, if $key(n_2) > key(n)$, 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, $key(n_2) \geq key(n_1)$. $\square$

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

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

There's nothing in between the node and its successor!

Proof.

Claim: 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 smaller keys than $key(n)$.

Claim: If the right subtree exists, then 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. □

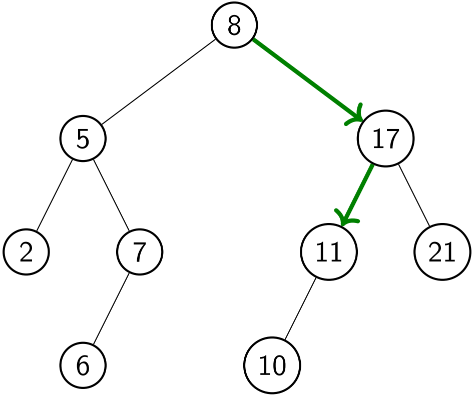
Exercise 6.7

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. Successor doesn't exist
b. Modify the above proof to support repeated elements in BST.

Example: Insert in BST

Example 6.8

Where do we insert 10?



Algorithm: Insert in BST

Algorithm 6.4: INSERT(BST T, Node n)

```
x := root(T);y := Null;
while x ≠ Null do
    y := x;
    if key(x) > key(n) then
        x := left(x)
    else
        x := right(x)
if y = Null then
    root(T) = n;
if key(y) > key(n) then
    left(y) := n
else
    right(y) := n
parent(n) = y
```

Exercise 6.8

a. What is the running time analysis of the algorithm?

b. Give an order of insertion when the height of a tree is maximum. such that it'll make skew tree.

c. Give an order of insertion when the height of a tree is minimum. Each insertion makes left and right subtree.

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,..

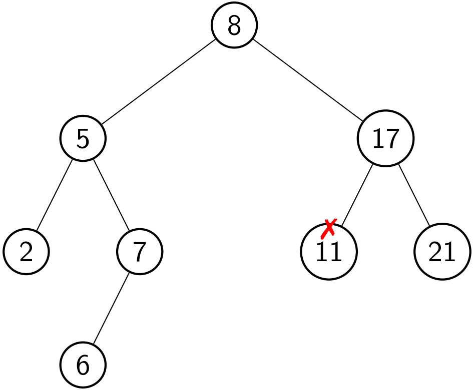
Topic 6.3

Deletion

Example: deleting a leaf

Example 6.9

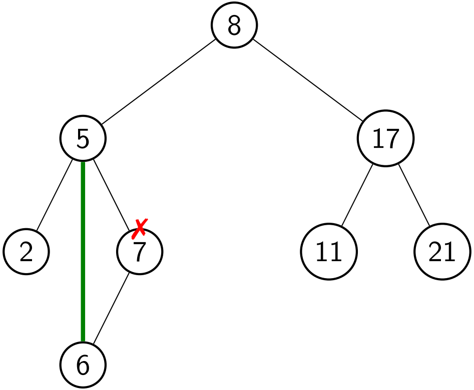
We delete leaf 11 *by simply removing the node.*



Example: deleting a node with a single child

Example 6.10

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

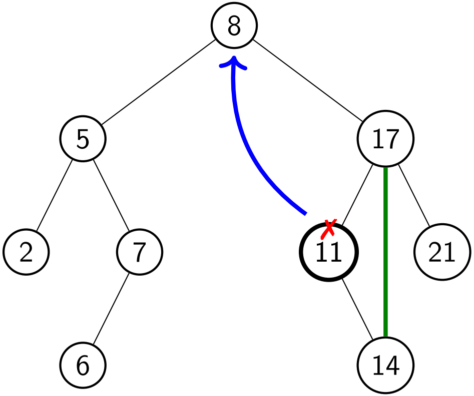


Example: deleting a node with both children

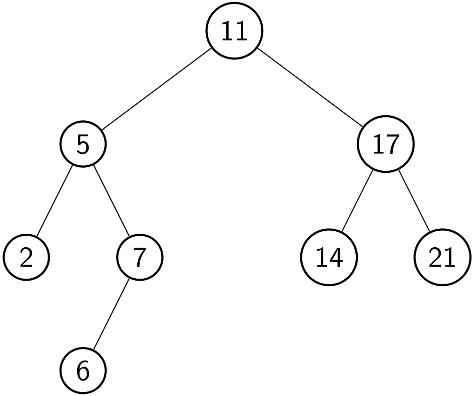
Here it becomes much clearer that why successor is important because we've to replace the element to delete with its successor

Example 6.11

We delete node 8 *by removing* 11, *which is the successor of* 8, *and then storing the data of* 11 *on* 8.



$\rightsquigarrow$



Algorithm: delete in BST

Doubt!

Algorithm 6.5: DELETE(BST T, Node n)

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

if $y \neq n$ **then** if y turns out to be a successor

  $key(n) := key(y)$ // 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

Topic 6.4

Average BST depth

Average cost of n-inserts

Let us consider a random permutation of $1, .., n$.

We insert the numbers in the order.

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.

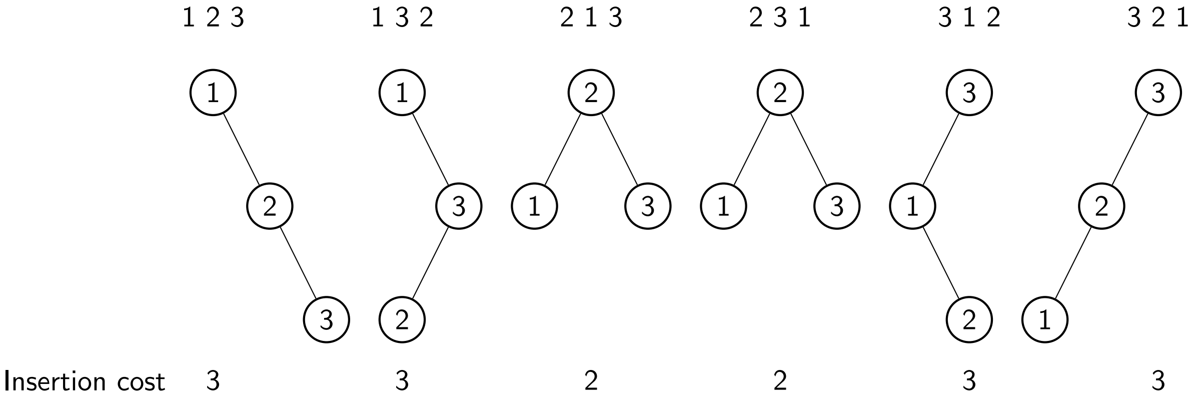
Exercise 6.9

What are the best and worst insertion times?

Example: Computing $T(n)$

Example 6.12

Let us compute the average cost of inserting three elements.



$T(3) = 16/6$

Recurrence for $T(n)$

In $(n-1)!$ permutations, i is the first element.

In 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)

There are $(i-1)!$ orderings for keys $1, ..., i-1$.

In the $(n-1)!$ permutations, each ordering of $(i-1)!$ occurs $(n-1)!/(i-1)!$.

If we only had keys $1, ..., i-1$, the average time is $T(i-1)$.

The total time to insert in all the orderings is $(i-1)!T(i-1)$.

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.

Therefore, the total time of insertion of $(i-1)!$ orderings is

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

Since each permutation occurs $(n-1)!/(i-1)!$, total time for insertions in the left subtree is

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

Similarly, total time for insertions in the right subtree is

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

Recurrence for $T(n)$(4)

The total time to insert all keys in the permutations where the first key is i is

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

Therefore, the total time of insertions in all permutations

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

Recurrence for $T(n)$(5)

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 $T(0) = 0$.

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

The relation for $n-1$.

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

After reordering the terms.

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

After reordering of terms in $T(n)$,

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

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

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

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

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

Commentary: $\int_1^n \frac{1}{x} dx = \ln n$

Expanding the approximate recurrence relation

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

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

Therefore,

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

Topic 6.5

Tutorial problems

Exercise: Sorting via BST

Exercise 6.10

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.

O(n) just do inorder traversal!

Exercise: delete all smaller keys

Exercise 6.11

Given a BST T and a key k, the task is to delete all keys $b < a$ from T. Write pseudocode to do this. How much time does your algorithm take? What is the structure of the tree left behind? What is its root?

Question is tricky one. Recursively delete the right and left subparts. Notice if the node at any level is outofrange then it does not imply that right or left subtree is also out of range.

Exercise: expected height

Exercise 6.12

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

Exercise: find leftmost and rightmost

Exercise 6.13

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

Topic 6.6

Problems

Exercise: post-order search tree

!hard

Exercise 6.14

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

hard!

Exercise 6.15

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

End of Lecture 6