Chapter 12
Binary Search Trees, etc.

*Binary Search trees* are data structures that support a variety of dynamic set operations, e.g., Search, Minimum, Maximum, Predecessors, Successors, Insert, and Delete. Thus, this *BST* data structure can be used to implement a dictionary and/or a priority queue (?).

An essential observation is that all these operations take time proportional to the height of the tree. Thus, for a *balanced search tree*, they all take $O(\log(n))$; ☺ but for a *degraded tree*, i.e., a linear list, they take $O(n)$. 😞

We will see that for a randomly built search tree, its expected height is $\Theta(\log n)$ through Project 6. We will also discuss some of the techniques to ensure the balance of such a tree so that we can achieve the ideal run time.
Why just binary tree?

The topological structure of a binary search tree is a binary tree, where each node could have at most two children, while in a general tree, this restriction is lifted.

A binary tree is certainly a general tree. On the other hand, we can convert any general tree into a binary tree, as follows:

Hence, these two classes of trees are equinumerous. We only need to study the simpler class of binary trees 😊.
Binary Search Trees

A binary search tree (BST) is a binary tree that may be empty.

A nonempty binary search tree has the following binary-search-tree property: 1) Every element has a key component, whose values are distinct. 2) The keys in the left subtree of the root are smaller than the key in the root. 3) The keys in the right subtree of the root are larger than that in the root. 4) Both the left and the right subtrees are binary search trees.

**Question:** Which of the following are BST(s)?

```
     20
    /  \
   15   25
  /  \\   /  \
 12 10 22
```

```
     30
    /  \
   5   40
    /   /  \
 2   65 70
```

```
     60
    /  \
   70
```

Since such a tree is usually not “nearly complete”, it is implemented in a linked structure.
Implementation

Below gives a node in a binary tree in *Java*:

```java
public class BinaryTreeNode {
    int key;
    BinaryTreeNode left, right, parent;

    public BinaryTreeNode(int key){
        this.key=key;
        left=right=parent=null;
    }
}
```

A binary tree looks like the following:

```java
public class BinaryTree {
    BinaryTreeNode root;

    public BinaryTree(){
        root=null;
    }
}
```

You will play with it in Project 6 to prepare for the *MoAP*.
Tree traversal

Traversal is an important operation, which systematically lists all the elements of a data structure.

It is trivial for a list (?), but can be implemented in various ways for a tree: pre-order, in-order, post-order, and level-order.

Each of them has its own applications, including evaluation of arithmetical and other expressions. (Cf. Lecture notes, Page 11-22, Functions and program structures, CS2470 System Programming in C/C++).

This important operation will be further discussed for the graph structure later on.

We explore yet another sorting algorithm, based on the in-order traversal of a binary tree.
Yet another sorting algorithm

Below is the *in-order traversal* algorithm, when applied to a binary tree.

\[
\text{IN-ORDER}(x)
\]
1. if \(x\)\(!=\)NIL
2. \text{IN-ORDER}(\text{left}[x])
3. \text{Visit}(\text{key}[x])
4. \text{IN-ORDER}(\text{right}[x])

**Claim:** If we apply an in-order traversal on a BST, we will get back a sorted list of its keys.

**Proof by strong induction on** \(n\): When the tree contains only one node, it is trivial.

Assume it is true for a tree containing less than \(n(\geq 1)\) nodes, we now prove it also holds for a tree with \(n\) nodes via strong induction.

We observe that both the left and right sub-trees contain less than \(n\) nodes(?).
The inductive case

When applying the in-order traversal to a tree, we apply it to the left-subtree first, since it contains strictly less than \( n(\geq 0) \) nodes, by the inductive assumption, this part will sort out all the nodes in the left-subtree.

The traversal algorithm then prints out the key of the root that, by definition of BST, is in the right place (?)

Finally, it goes to the right-subtree. By the same token, this second part also sorts out all the keys in the right-subtree.

Thus, the traversal indeed correctly sorts out all the keys 😊.

Is this similar to the argument for Quicksort?
This sorting is linear

Through an analysis similar to what we did for the Quicksort, we can also show (Check out the following few pages, Page 8 through 10), it takes $\Theta(n)$ time, on average, to traverse a binary tree in an in-order way.

**Question:** Have we found a general $\Theta(n)$ comparison based sorting algorithm?

**No:** As we will see, it takes $\Omega(n \log n)$ to set up a BST for $n$ elements, since the minimum height of a “balanced” tree with $n$ numbers is at least $\log n$.

**Homework:** Exercises 12.1-1 and 12.1-2.
Why is this sorting linear?

**Theorem 12.1.** If $x$ is the root of an $n$–node subtree, then it takes $\Theta(n)$ on average to traverse a binary tree in an in-order way.

**Proof:** Let $T(n)$ stand for the time in question. The procedure takes a small constant time to process an empty tree. In general, assume it applies to a node $x$, whose left subtree has $k$ nodes, thus its right one has $n - 1 - k$ nodes, where $k \in [0, n - 1]$.

Assume it takes $c$ (respectively, $d$) units of time to check a pointer (respectively, “visit $x$”), we have the following recurrence:

$$
T(0) = c
$$
$$
T(n) = \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n - 1 - k) + d]
$$
$$
\approx \frac{2}{n} \sum_{k=0}^{n-1} T(k) + d.
$$
From

\[ nT(n) = 2 \sum_{k=0}^{n-1} T(k) + dn. \] (1)

we can similarly get the following:

\[(n-1)T(n-1) = 2 \sum_{k=0}^{n-2} T(k) + d(n-1). \] (2)

We subtract Eq. 2 from Eq. 1 to derive

\[
\begin{align*}
&nT(n) - (n-1)T(n-1) \\
&= 2 \left[ \sum_{k=0}^{n-1} T(k) - \sum_{k=0}^{n-2} T(k) \right] + d(n-n+1) \\
&= 2T(n-1) + d.
\end{align*}
\]

Now, we have

\[ nT(n) = (n+1)T(n-1) + d. \]

Divide both sides by \(n(n+1)\), we get the following:

\[
\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{d}{n(n+1)}.
\]
Wrap up...

We can now apply the telescope technique (Cf. Page 8 of the Math preview notes).

\[
\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{d}{n(n+1)}
\]

\[
= \left[ \frac{T(n-2)}{n-1} + \frac{d}{(n-1)n} \right] + \frac{d}{n(n+1)}
\]

\[
= \cdots
\]

\[
= T(0) + d \sum_{k=1}^{n} \frac{1}{k(k+1)}
\]

Therefore,

\[
T(n) = c(n+1) + dn = (c+d)n + c
\]

\[
= \Theta(n).
\]

We are done. 😊
The search operation

This is the name of the game.

When looking for a key value $k$ in a binary search tree $T$, we begin at the top. If the root of $T$ is $NIL$, $k$ cannot be anywhere in $T$. Thus, we have failed.

Otherwise, we compare $k$ with the key of the root. There can be only three possible outcomes.

If they are equal, we are done. If $k$ is larger, it must be in the right subtree, if it is inside the tree at all; so, we recursively search for $k$ in the right subtree. Otherwise, we search the left one.
The code

We would call the following function with TREE-SEARCH(root[T], k).

TREE-SEARCH(x, k)
1. if x=NIL or k=key[x]
2. then return x
3. if k<key[x]
4. then return TREE-SEARCH(left[x],k)
5. else return TREE-SEARCH(right[x],k)

We can easily (?) write a non-recursive search procedure.

**Question:** Why is recursion expensive?

Obviously, the search can be done in $O(h)$, where $h$ is the height of the tree.

**Question:** Does it remind us of binary search?
Minimum and maximum

By the very structure of a $\text{BST}$, anything less than the key in a node is on its left subtree. Hence, the minimum key has to be in the left subtree of everything, and itself can’t have a left child. (Minimum in a maxHeap...) This observation leads to the following procedure.

TREE-MINIMUM(x)
1. If $x=$NIL
2. then return NIL
3. else
4. while left($x$)!=$\text{NIL}$
5. do $x<-$left[$x$]
6. return $x$

The search can be done in $O(h)$, where $h$ is the height of the tree.

The procedure for finding the maximum is symmetric.
Successor and predecessor

Given a node, $x$, in a BST, its successor is the next node in the in-order sequence of all the nodes in the tree. In other words, the successor of $x$ is the smallest node among all those larger than $x$.

We need to consider two cases: If the right sub-tree of $x$ exists, i.e., if $\text{right}[x] \neq \text{NIL}$, its successor is simply the minimum of the subtree rooted at $\text{right}[x]$, as we just went through.

When $\text{right}[x]$ is empty, $x$ has to be the maximum element of the “minimum tree”, where it is located. Thus, none of the elements in this minimum tree could be bigger, thus its successor.

In fact, the successor of $x$ has to be the parent, $y$, of $r$, the root of such a tree, if it exists. And $r$ has to be the left child of $y$. (?)

An example might help... .
How to get it?

Technically, in the former case, we return the minimum of right(x); and in the latter case, we return the parent of a left child which is the root of the minimum tree that contains x.

TREE-SUCCESSOR(x)
1. if right(x)\neq\text{NIL}
2. then return TREE-MINIMUM(right(x)))
3. y\leftarrow p[x]
4. while y\neq\text{NIL} and x=right[y]
5. do x\leftarrow y
6. \quad y\leftarrow p[y]
7. //Notice that y could be NIL if the minimum tree is the whole tree
8. //minimum tree is the whole tree
9. return y

The successor, and the similar predecessor, procedures also run in $O(h)$. 

Homework: Exercise 12.2-1
The insertion operation

To insert an element with key \( k \) into a BST, we first verify that this key does not occur anywhere inside the tree by doing a search.

If the search succeeds, the insertion will not be done, or maybe a frequency field is updated; otherwise, the element will be inserted where the search reports a failure.

The following shows an example, where we add in 13 into a BST.

![BST Diagram]

The procedure also takes \( O(h) \), where \( h \) is the height of the tree.
The code

Below is a *non-recursive version*, called with TREE-INSERT(root[T], z), where T is a *BST* and z refers to a node to be inserted into T.

TREE-INSERT(x, z)
1. y<-NIL  //refers to parent of x
2. while x!=NIL
3. do y<-x  //y is the parent of x
4. if key[z]<key[x]
5. x<-left[x]
6. else x<-right[x]
7. p[z]<-y
8. if y=NIL  //empty tree
9. root[T]<-z
10. else if key[z]<key[y]
11. left[y]<-z
12. else right[y]<-z

**Question:** How about a recursive version? Exercise 12.3-1.
A bit more details

In terms of its implementation, put the value into a node $z$, we look for the insertion point, starting at the root, based on the BST property, as indicated by $x$ in the code, and use $y$ to keep track of its parent.

When the insertion point is found, if the tree is empty, when $y$ is NIL, we directly add in the node as the only one in the tree. Otherwise, depending on the relationship between the keys in $z$ and $y$, we will add in $z$ as either the left or the right child of $y$, as $y$ is always set to be the parent of $z$.

Here the reference $y$ plays a role similar to the prev reference in the doubly linked list, so we don’t need to completely hunt down the parent.

**Homework:** Exercises 12.3-1 and 12.3-3.
The deletion operation

This one is the pain...😊. If the node to be deleted is a leaf, we simply delete it. It is also easy to do when it has just one child.

When it has two children, it becomes a bit tricky, since we have only one location for two nodes to be hooked on.
The code

TREE-DELETE(T, z)
1. if left[z]=NIL or right[z]=NIL
2. y<-z //At most one child
3. else y<-TREE-SUCCESSOR(z) //y has <= 1 child
4. if left[y]!=NIL
5. x<-left[y]
6. else x<-right[y] //x is y’s child
7. if x!=NIL //Set up parent of x
8. p[x]<-p[y]
9. if p[y]=NIL //Deleting the only node
10. root[T]<-x
11. else //Set up the correct child
12. if y=left[p[y]]
13. left[p[y]]<-x
14. else right[p[y]]<-x
15. if y!=z //When z has two children
16. //copy y’s data into z
17. key[z]<-key[y]
18. return y
A bit more details

When deleting a node \( z \) from the tree, we first decide if \( z \) has no children, just one, or two children.

In the first two cases, we can set \( z \) to \( y \), the one to be deleted. In the last case, we set \( y \) to be the successor of \( z \) (should we?) and will copy over its value to the node where \( z \) locates, and delete \( y \) instead. Note that in this case, \( y \) will have its left child empty (why?). Now, \( y \) has at most one (right) child (Cf. Exercise 12.3-6).

We then set \( x \) to point to this child, and delete \( y \) in line 7, by adjusting the parent pointer of \( x \) to be that of \( y \). We finally set \( x \) to be a child of \( y \)'s parent.

**Homework:** Exercise 12.3-6
If insertion and deletions are made at random, then the height of such a BST will be $O(\log n)$, which leads to $O(n \log n)$ to build a BST. Indeed, if we randomly insert $n$ keys into an initially empty BST to obtain a randomly built BST, we get the following result:

**Theorem 12.4.** The expected height of a randomly built BST on $n$ keys is $O(\log n)$.

**Argument via experiment:** Project 6.

On the other hand, if we use the original insertion function to add in element 1, 2, ..., $n$, in an empty BST, then the resulted BST is actually a linear list.

As a result, a basic operation can take as much as $O(n)$. 😊
Balanced trees

It really means that the left and right sides of a tree, and that of its subtrees, is of more or less the same height.

We saw that the height of a complete binary tree with $n$ nodes is $\log_2(n + 1)$, and that of a nearly complete binary tree is also in $\Theta(\log n)$.

But a complete binary tree can only be constructed for a special class of numbers when $n = 2^h - 1$, where $h \geq 1$, i.e., 1, 3, 7, 15,...; and there is also a structural restriction to all the nearly complete binary trees.

Thus, we have to come up with a more generous structure, while making sure the height is still in the order of $\log n$.

AVL trees meet our needs pretty well.
What do we want?

We want to define the BST operations in such a way that the trees stay balanced all the time, while preserving the binary search tree properties.

The essential algorithms for insertion and deletion will be exactly the same as for BST, except that, as they might destroy the property of being balanced, we have to restore the balance of the tree, if needed. It is guaranteed that all the basic operations applied to a balanced tree will be done in $\Theta(\log n)$.

The class of AVL trees is one of the more popular balanced trees. An nonempty binary tree with $T_L$ and $T_R$ as its left and right subtrees is an AVL tree if and only if both $T_L$ and $T_R$ are AVL trees, and $|h_L - h_R| \leq 1$.

For another class, the Black and Red trees, check out Chapter 13 of the textbook.
AVL search trees

An AVL search tree is a binary search tree such that

1. The height of an AVL tree with \( n \) nodes is \( O(\log n) \). (Page 26 through 28)

2. For every \( n \geq 0 \), there exists an AVL tree with \( n \) nodes. 😊

3. We can search an \( n \)-element AVL search tree in \( O(\log n) \). (Because of Item 1))

4. A new element can be added into an \( n \)-element AVL tree, in \( O(\log n) \), so that the resulted \( (n+1) \)-element tree is also an AVL tree; and

5. an element can be deleted from an \( n \)-element AVL search tree, in \( O(\log n) \), and the resulted \( (n-1) \)-element tree is also an AVL tree. (Page 29 through 31)
Fibonacci tree

**Question:** What is the least number of nodes in a balanced binary tree with height $h$?

Let $S(h)$ be this number. Obviously, $S(0) = 1$ and $S(1) = 2$. Notice a balanced tree with height 1 could contain either two or three nodes.

In general, we can construct such a balanced tree with height $h$ by merging two subtrees of heights $h - 1$ and $h - 2$, each of which has the least number of nodes.

We call such trees *Fibonacci trees* for the obvious reason.

Now what?
A bit math.

Assume that for all $1 \leq h' < h$, $S(h') \geq fib(h')$.

$$\begin{align*}
S(h) &= S(h - 1) + S(h - 2) + 1 \\
&\geq fib(h - 1) + fib(h - 2) + 1 \\
&= fib(h) + 1 > fib(h).
\end{align*}$$

Hence, for all $h \geq 1$, $S(h) \geq fib(h)$.

Moreover, assume for all $1 \leq h' < h$, $fib(h') \geq \left(\frac{3}{2}\right)^{h' - 1}$, then

$$\begin{align*}
fib(h) &= fib(h - 1) + fib(h - 2) \\
&\geq \left(\frac{3}{2}\right)^{h - 2} + \left(\frac{3}{2}\right)^{h - 3} \\
&= \left(\frac{3}{2}\right)^{h - 1} \times \left[\frac{2}{3} + \frac{4}{9}\right] \\
&= \frac{10}{9} \left(\frac{3}{2}\right)^{h - 1} > \left(\frac{3}{2}\right)^{h - 1}.
\end{align*}$$

Therefore, for all $h \geq 1$, $S(h) \geq \left(\frac{3}{2}\right)^{h - 1}$. 
Height of an AVL tree

Given an AVL tree with \( n \) nodes of height \( h (\geq 1) \), we have that

\[
n \geq S(h) \geq \left( \frac{3}{2} \right)^{h-1}
\]

Further calculation leads to the following result,

\[
h \leq \frac{\log n}{\log 3 - 1} + 1 = \Theta(\log n).
\]

A simpler, but yet proved, result is that

\[
h \leq \log(n + 1) + 0.25.
\]

This takes care of the height of an AVL tree, i.e., Item 1, thus, also Item 3. Now, we move to Items 4 and 5, where we will show that, after an insertion and deletion, how to restore the balance, if necessary, which will take care of Item 2.
Represent an AVL tree

An AVL tree is usually represented using the linked representation for binary trees.

To provide information to restore the balance, a balance factor is added to each node. For any node, this factor is defined to be the difference between the heights of its left and right subtrees.

Question: Why the balance factor of the node containing 25 in the tree on the left is “-1”?

It is easy to see that any search takes $O(\log n)$. 
Restore the Balance (I)

When a tree becomes unbalanced, as a result of either insertion or deletion, we can restore its balance by rotating the tree at and/or below the node where the imbalance is detected.

**Case 1:** Assume that the left subtree rooted at \( k_1 \) has height larger than that of \( Z \), the right subtree of the one rooted at \( k_2 \), plus 1, caused by deleting a node from \( Z \), or inserting a node into the one rooted at \( k_1 \). We apply a left rotation, which will restore the balance of the tree rooted at \( k_1 \).

Notice that this rule is symmetric.
Case 2: Suppose that the imbalance is caused by the right subtree rooted at $k_1$ being too high, whose balance can’t be restored by a single rotation. We must apply two successive single rotations, combined into the following (R-L) double rotation rule.

Similarly, if the left subtree is too high, we can apply the following (L-R) double rotation.
Two examples

Below shows an application of single rotation rule to adjust the balance, also demonstrating Item 2.

Similarly, the following shows an application of a (R-L) double rotation rule.
Histogramming

We start with a collection of $n$ keys and want to output a list of distinct keys and their frequencies. Assume that $n = 10$, and $keys = [2, 4, 2, 2, 3, 4, 2, 6, 4, 2]$.

<table>
<thead>
<tr>
<th>key</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

**Question:** How to do it?

When the range of those keys is quite small, this problem can be solved very easily, in $\Theta(n)$, by using an array $h[0..r]$, where $r$ is the maximum key value, and letting $h[i]$ store the frequency of $i$. 
More general cases

When the key values are not integers, we can sort the \( n \) keys, in \( \Theta(n \log n) \), then scan the sorted list from left to right to find the frequencies of respective keys.

This solution can be further improved, when \( m \), the number of distinct keys, is quite small. We can use a (balanced) BST tree, each of whose nodes contains two fields, key and frequency. We insert all the \( n \) keys into such a tree, when a key is already there, we simply increment its frequency by 1.

This procedure takes an expected complexity of \( \Theta(n \log m) \). When a balanced tree is used, this complexity is guaranteed, even in the worst case.

We need this stuff for Project 7.