Binary Search Trees
Dictionary (Map) ADT

An dictionary is an ADT that allows us to store a collection of records of the form

(key, data)

and it provides the following operations: get (key), put (key, data), remove(key)
Ordered Dictionary (Map) ADT

An ordered dictionary is an ADT that allows us to store a collection of records of the form 

\[(\text{key, data})\]

where the key attributes come from a total order.
Total Order

The keys stored in an ordered dictionary must come from a **total order**: given any two keys $k$ and $k'$ we can always determine whether

- $k = k'$
- $k > k'$, or
- $k < k'$

For example total orders are defined over the integers, real numbers, characters, Strings, ...
Ordered Dictionary (Map) ADT

- get (k): record with key k
- put (k, data): add record (k, data)
- remove (k): delete record with key k
- smallest(): record with smallest key
- largest(): record with largest key
- predecessor(k): record with largest key less than k
- successor(k): record with smallest key greater than k
Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array sorted by key

- Performance:
  - Searches take $O(\log n)$ time, using binary search
  - Inserting a new item takes $O(n)$ time, since in the worst case we have to shift $n - 1$ items to make room for the new item
  - Removing an item takes $O(n)$ time, since in the worst case we have to shift $n - 1$ items to compact the items after the removal

- The search table is effective only for ordered maps of small size or for maps on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)
Binary Search Trees

A binary search tree is a proper binary tree storing records in its internal nodes such that for each internal node $u$:

- Every key in the left subtree of $u$ is smaller than $key(u)$
- Every key in the right subtree of $u$ is larger than $key(u)$.
Binary Search Trees

A binary search tree is a **proper** binary tree **storing records** in its internal nodes such that for each internal node $u$:
- Every key in the left subtree of $u$ is smaller than $key(u)$
- Every key in the right subtree of $u$ is larger than $key(u)$.

Leaves **do not** store records
A binary search tree is a proper binary tree storing records in its internal nodes such that for each internal node $u$:

- Every key in the left subtree of $u$ is smaller than $key(u)$
- Every key in the right subtree of $u$ is larger than $key(u)$.

Leaves do not store records.
A binary search tree is a proper binary tree storing records in its internal nodes such that for each internal node $u$:
- Every key in the left subtree of $u$ is smaller than $key(u)$
- Every key in the right subtree of $u$ is larger than $key(u)$.
Leaves do not store records
Binary Search Trees

A binary search tree is a proper binary tree storing records in its internal nodes such that for each internal node \( u \):

- Every key in the left subtree of \( u \) is smaller than \( \text{key}(u) \).
- Every key in the right subtree of \( u \) is larger than \( \text{key}(u) \).

Leaves do not store records.
Inorder Traversal

An inorder traversal of a binary search trees visits the keys in increasing order

1, 2, 4, 6, 8, 9
Ordered Dictionary (Map) ADT

- get (k): record with key k
- put (k,data): add record (k,data)
- remove (k): delete record with key k
- smallest(): record with smallest key
- largest(): record with largest key
- predecessor(k): record with largest key less than k
- successor(k): record with smallest key greater than k
Algorithm smallest (r)
In: Root r of a binary search tree
Out: Node storing smallest key, or null if the tree has no data in it

\[ C_2 \begin{cases} 
\text{if } r \text{ is a leaf then return null} \\
\text{else } \\
\quad p \leftarrow r \\
\quad \text{while } p \text{ is an internal node do} \\
\quad \quad p \leftarrow \text{left child of } p \\
\quad \text{return parent of } p 
\end{cases} \]

\[ C_1 \{ \]

\# iterations = \# nodes in leftmost branch
\[ \leq \text{height} + 1 \]

\[ f(n) = C_2 + C_1 (\text{height} + 1) \]

is \( O(\text{height}) \)
Algorithm get \( (r, k) \)

In: Root \( r \) of a binary search tree, key \( k \).

Out: Node storing \( k \), or leaf where \( k \) should have been stored.

\[
\begin{align*}
\text{if } r \text{ is a leaf} & \quad \text{then return } r \\
\text{else} & \\
\text{if } k = \text{key stored in } r & \quad \text{then return } r \\
\text{else } k < \text{key stored in } r & \quad \text{return } \text{get(left child of } r, k) \\
\text{else} & \quad \text{return } \text{get(right child of } r, k)
\end{align*}
\]

How many calls?

Worst case: \( k \) not in tree

At most height + 1 calls, so

\( f(n) = \mathcal{O}(\text{height} + 1) \)

is \( \mathcal{O}(\text{height}(n)) \)
Algorithm put (r, k, d)
In: Root r of a binary search tree, record (k, d)
Out: True if (k, d) was added to the tree, false otherwise

O(height) \{ p \leftarrow \text{get} (r, k) \\
\text{if} \ p \ \text{is internal node} \ \text{then} \\
\quad \text{return} \ \text{false} \\
\text{else} \ \text{create} \ 2 \ \text{leaves and set them as children of} \ p \\
\quad p.\text{key} \leftarrow k \\
\quad p.\text{data} \leftarrow d \\
\quad \text{return} \ \text{true} \}
Algorithm remove \((r, k)\)

In: Root \(r\) of a binary search tree, key \(k\)

Out: True if \(k\) was removed, false otherwise

\[
\text{Time complexity } f(n) \text{ is } \mathcal{O}(\text{height})
\]
Ordered Dictionary (Map) ADT

- get (k): record with key k
- put (k, data): add record (k, data)
- remove (k): delete record with key k
- smallest(): record with smallest key
- largest(): record with largest key
- predecessor(k): record with largest key less than k
- successor(k): record with smallest key greater than k
Algorithm successor \((r, k)\)

In: Root \(r\) of a binary search tree, key \(k\)

Out: Node storing the successor of \(k\), or null if \(k\) has no successor

if \(r\) is a leaf then return null
else {
    \(p \leftarrow \text{get} (r, k)\) \(\{\text{O(height)}\}\)
    if \((p\) is an internal node) and \((\text{right child of } p\) is internal\) then
        return smallest (right child of \(p\))
    else {
        \(p' \leftarrow \text{parent of } p\)
        while \((p' \neq r)\) and \((p' \text{ is the right child of } p)\) do
            \(p' \leftarrow \text{parent of } p'\)
        if \(p = r\) then return null
        else return \(p'\)
    }
}
Search

To search for a key $k$, we trace a downward path starting at the root.

The next node visited depends on the comparison of $k$ with the key of the current node.

If we reach a leaf, the key is not found.

Example: $\text{get}(4)$:
- Call $\text{TreeSearch}(4, \text{root})$

```
Algorithm $\text{TreeSearch}(k, v)$
    if $v$ is a leaf then
        return $v$
    if $k < \text{key}(v)$ then
        return $\text{TreeSearch}(k, \text{left}(v))$
    else if $k = \text{key}(v)$ then
        return $v$
    else 
        $k > \text{key}(v)$
    return $\text{TreeSearch}(k, \text{right}(v))$
```
Insertion

To perform operation put(k, δ), we search for key k (using TreeSearch)
Assume k is not already in the tree, and let w be the leaf reached by the search
We insert k at node w and expand w into an internal node
Example: insert 5
Deletion

To perform operation \( \text{remove}(k) \), we search for key \( k \)

Assume key \( k \) is in the tree, and let \( v \) be the node storing \( k \)

If node \( v \) has a leaf child \( w \), we remove \( v \) and \( w \) from the tree with operation \( \text{removeExternal}(w) \), which removes \( w \) and its parent

Example: remove 4
Deletion (cont.)

- We consider the case where the key $k$ to be removed is stored at a node $v$ whose children are both internal
  - we find the internal node $w$ that follows $v$ in an inorder traversal
  - we copy $key(w)$ into node $v$
  - we remove node $w$ and its left child $z$ (which must be a leaf) by means of operation $\text{removeExternal}(z)$

Example: remove 3
Performance

- Consider an ordered dictionary with $n$ items implemented by means of a binary search tree of height $h$
  - the space used is $O(n)$
  - methods get, put and remove take $O(h)$ time

- The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case