7.1 Binary Search Trees

An (internal) binary search tree stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node \( v \) have a smaller key-value than \( \text{key}[v] \) and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:
7.1 Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- $T.\text{insert}(x)$
- $T.\text{delete}(x)$
- $T.\text{search}(k)$
- $T.\text{successor}(x)$
- $T.\text{predecessor}(x)$
- $T.\text{minimum}()$
- $T.\text{maximum}()$
Algorithm 1 TreeSearch(x, k)

1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Binary Search Trees: Searching

TreeSearch(root, 17)

Algorithm 1 TreeSearch(x, k)
1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Binary Search Trees: Searching

TreeSearch(root, 17)

Algorithm 1 TreeSearch(x, k)
1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Algorithm 1 \textbf{TreeSearch}(x, k)

\begin{enumerate}
\item \textbf{if} $x = \text{null}$ \textbf{or} $k = \text{key}[x]$ \textbf{return} $x$
\item \textbf{if} $k < \text{key}[x]$ \textbf{return} \textbf{TreeSearch}(\text{left}[x], k)$
\item \textbf{else return} \textbf{TreeSearch}(\text{right}[x], k)$
\end{enumerate}
Algorithm 1 TreeSearch($x, k$)

1: if $x = \text{null}$ or $k = \text{key}[x]$ return $x$
2: if $k < \text{key}[x]$ return TreeSearch(left[$x$], $k$)
3: else return TreeSearch(right[$x$], $k$)
Binary Search Trees: Searching

TreeSearch(root, 17)

Algorithm 1 TreeSearch(x, k)

1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Algorithm 1 TreeSearch($x, k$)

1. if $x = \text{null}$ or $k = \text{key}[x]$ return $x$
2. if $k < \text{key}[x]$ return TreeSearch(left[$x$], $k$)
3. else return TreeSearch(right[$x$], $k$)
Algorithm 1 TreeSearch(x, k)
1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Binary Search Trees: Searching

TreeSearch(root, 8)

Algorithm 1 TreeSearch(x, k)

1: if \( x = \text{null} \) or \( k = \text{key}[x] \) return \( x \)
2: if \( k < \text{key}[x] \) return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Algorithm 1 TreeSearch\( (x, k) \)

1. \textbf{if} \( x = \text{null} \) \textbf{or} \( k = \text{key}[x] \) \textbf{return} \( x \)
2. \textbf{if} \( k < \text{key}[x] \) \textbf{return} TreeSearch(left[x], k)
3. \textbf{else return} TreeSearch(right[x], k)
Algorithm 1 TreeSearch\((x, k)\)

1. if \(x = \text{null} \) or \(k = \text{key}[x]\) return \(x\)
2. if \(k < \text{key}[x]\) return TreeSearch\((\text{left}[x], k)\)
3. else return TreeSearch\((\text{right}[x], k)\)
Binary Search Trees: Searching

**TreeSearch(root, 8)**

```
Algorithm 1 TreeSearch(x, k)
1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
```
Algorithm 1 TreeSearch(x, k)
1: if x = null or k = key[x] return x
2: if k < key[x] return TreeSearch(left[x], k)
3: else return TreeSearch(right[x], k)
Binary Search Trees: Searching

Algorithm 1

TreeSearch(x, k)

1. if x = null or k = key[x] return x
2. if k < key[x] return TreeSearch(left[x], k)
3. else return TreeSearch(right[x], k)
Algorithm 2 \text{TreeMin}(x)
\begin{itemize}
\item[1:] \textbf{if } x = \text{null} \textbf{ or left}[x] = \text{null} \textbf{ return } x
\item[2:] \textbf{return} \text{TreeMin}(\text{left}[x])
\end{itemize}
**Algorithm 2** \( \text{TreeMin}(x) \)

1. \( \textbf{if} \ x = \text{null or left}[x] = \text{null} \ \textbf{return} \ x \)
2. \( \textbf{return} \ \text{TreeMin}(\text{left}[x]) \)
Algorithm 2 TreeMin(x)
1: if x = null or left[x] = null return x
2: return TreeMin(left[x])
Algorithm 2 TreeMin(x)

1: if x = null or left[x] = null return x
2: return TreeMin(left[x])
Algorithm 2 TreeMin($x$)

1: if $x$ = null or left[$x$] = null return $x$
2: return TreeMin(left[$x$])
Algorithm 2 TreeMin(x)
1: if x = null or left[x] = null return x
2: return TreeMin(left[x])
Binary Search Trees: Successor

Algorithm 3 TreeSucc(x)

1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
4: x ← y; y ← parent[x]
5: return y;
Algorithm 3  TreeSucc($x$)

1: if right[$x$] ≠ null return TreeMin(right[$x$])
2: $y ←$ parent[$x$]
3: while $y$ ≠ null and $x = $ right[$y$] do
4: $x ← y$; $y ←$ parent[$x$]
5: return $y$;
Algorithm 3 TreeSucc(x)

1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
   4: x ← y; y ← parent[x]
5: return y;
Algorithm 3 TreeSucc(\(x\))

1. if right[\(x\)] \(\neq\) null return TreeMin(right[\(x\)])
2. \(y \leftarrow\) parent[\(x\)]
3. while \(y \neq\) null and \(x =\) right[\(y\)] do
4. \(x \leftarrow y; y \leftarrow\) parent[\(x\)]
5. return \(y\);
Algorithm 3 TreeSucc(x)

1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
4: x ← y; y ← parent[x]
5: return y;
Algorithm 3 TreeSucc(x)
1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
4: x ← y; y ← parent[x]
5: return y
Algorithm 3 TreeSucc(x)

1: if right[x] ≠ null return TreeMin(right[x])
2: y ← parent[x]
3: while y ≠ null and x = right[y] do
4:   x ← y; y ← parent[x]
5: return y;
Binary Search Trees: Insert

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

**Algorithm 4** 

`TreeInsert(x, z)`

1: if $x = \text{null}$ then  
2: \quad \text{root}[T] \leftarrow z; \text{parent}[z] \leftarrow \text{null};  
3: \quad \text{return};  
4: if $\text{key}[x] > \text{key}[z]$ then  
5: \quad if $\text{left}[x] = \text{null}$ then  
6: \quad \quad \text{left}[x] \leftarrow z; \text{parent}[z] \leftarrow x;  
7: \quad \quad \text{else} \ TreeInsert(\text{left}[x], z);  
8: \quad \text{else}  
9: \quad \quad if \ \text{right}[x] = \text{null} \text{ then}  
10: \quad \quad \quad \text{right}[x] \leftarrow z; \text{parent}[z] \leftarrow x;  
11: \quad \quad \text{else} \ TreeInsert(\text{right}[x], z);
Binary Search Trees: Insert

Insert element not in the tree.

Algorithm 4 TreeInsert(x, z)

1: if x = null then
2: root[T] ← z; parent[z] ← null;
3: return;
4: if key[x] > key[z] then
5: if left[x] = null then
6: left[x] ← z; parent[z] ← x;
7: else TreeInsert(left[x], z);
8: else
9: if right[x] = null then
10: right[x] ← z; parent[z] ← x;
11: else TreeInsert(right[x], z);
Binary Search Trees: Insert

Insert element not in the tree.

Search for $z$. At some point the search stops at a null-pointer. This is the place to insert $z$.

Algorithm 4 TreelInsert($x, z$)

1: if $x = \text{null}$ then
2: root[$T$] ← $z$; parent[$z$] ← null;
3: return;
4: if key[$x$] > key[$z$] then
5: if left[$x$] = null then
6: left[$x$] ← $z$; parent[$z$] ← $x$;
7: else TreelInsert(left[$x$], $z$);
8: else
9: if right[$x$] = null then
10: right[$x$] ← $z$; parent[$z$] ← $x$;
11: else TreelInsert(right[$x$], $z$);
Binary Search Trees: Insert

Insert element **not** in the tree.

**TreelInsert**(root, 20)

Search for $z$. At some point the search stops at a null-pointer. This is the place to insert $z$.

**Algorithm 4 TreelInsert**(x, z)

1: if $x = \text{null}$ then
2: root[$T$] $\leftarrow$ z; parent[z] $\leftarrow$ null;
3: return;
4: if key[x] $>$ key[z] then
5: if left[x] = null then
6: left[x] $\leftarrow$ z; parent[z] $\leftarrow$ x;
7: else TreelInsert(left[x], z);
8: else
9: if right[x] = null then
10: right[x] $\leftarrow$ z; parent[z] $\leftarrow$ x;
11: else TreelInsert(right[x], z);
Binary Search Trees: Insert

Insert element **not** in the tree.

**TreeInsert**(root, 20)

Search for \( z \). At some point the search stops at a null-pointer. This is the place to insert \( z \).

**Algorithm 4** TreeInsert\((x, z)\)

1: if \( x = \text{null} \) then
2: \( \text{root}[T] \leftarrow z; \text{parent}[z] \leftarrow \text{null}; \)
3: return;
4: if key\([x]\) > key\([z]\) then
5: if left\([x]\) = null then
6: left\([x]\) \leftarrow z; parent\([z]\) \leftarrow x;
7: else TreeInsert(left\([x]\), z);
8: else
9: if right\([x]\) = null then
10: right\([x]\) \leftarrow z; parent\([z]\) \leftarrow x;
11: else TreeInsert(right\([x]\), z);
Binary Search Trees: Insert

Insert element not in the tree.

TreeInsert(root, 20)

Search for $z$. At some point the search stops at a null-pointer. This is the place to insert $z$.

Algorithm 4 TreeInsert($x, z$)

1: if $x = \text{null}$ then
2:    root[$T$] $\leftarrow z$; parent[$z$] $\leftarrow \text{null}$;
3:    return;
4: if key[$x$] $>$ key[$z$] then
5:    if left[$x$] = null then
6:        left[$x$] $\leftarrow z$; parent[$z$] $\leftarrow x$;
7:    else TreeInsert(left[$x$], $z$);
8: else
9:    if right[$x$] = null then
10:       right[$x$] $\leftarrow z$; parent[$z$] $\leftarrow x$;
11: else TreeInsert(right[$x$], $z$);
Binary Search Trees: Insert

Insert element **not** in the tree.

**TreeInsert**(root, 20)

Search for \( z \). At some point the search stops at a null-pointer. This is the place to insert \( z \).

**Algorithm 4 TreeInsert**(*x, z*)

1: **if** \( x = \text{null} \) **then**
2: \( \text{root}[T] \leftarrow z; \text{parent}[z] \leftarrow \text{null}; \)
3: **return**;
4: **if** \( \text{key}[x] > \text{key}[z] \) **then**
5: **if** \( \text{left}[x] = \text{null} \) **then**
6: \( \text{left}[x] \leftarrow z; \text{parent}[z] \leftarrow x; \)
7: **else** TreeInsert(left[x], z);
8: **else**
9: **if** \( \text{right}[x] = \text{null} \) **then**
10: \( \text{right}[x] \leftarrow z; \text{parent}[z] \leftarrow x; \)
11: **else** TreeInsert(right[x], z);
Binary Search Trees: Insert
Insert element **not** in the tree.

\textbf{TreelInsert}(\texttt{root, 20})

![Binary Search Tree Diagram]

Search for \textit{z}. At some point the search stops at a null-pointer. This is the place to insert \textit{z}.

**Algorithm 4 TreelInsert(\textit{x, z})**

1: if \textit{x} = null then
2: \hspace{1em} root[\textit{T}] \leftarrow \textit{z}; parent[\textit{z}] \leftarrow \text{null};
3: \hspace{1em} return;
4: if key[\textit{x}] > key[\textit{z}] then
5: \hspace{2em} if left[\textit{x}] = null then
6: \hspace{3em} left[\textit{x}] \leftarrow \textit{z}; parent[\textit{z}] \leftarrow \textit{x};
7: \hspace{2em} else TreelInsert(left[\textit{x}], \textit{z});
8: \hspace{1em} else
9: \hspace{2em} if right[\textit{x}] = null then
10: \hspace{3em} right[\textit{x}] \leftarrow \textit{z}; parent[\textit{z}] \leftarrow \textit{x};
11: \hspace{2em} else TreelInsert(right[\textit{x}], \textit{z});
Binary Search Trees: Insert

Insert element not in the tree.

TreeInsert(root, 20)

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Algorithm 4 TreeInsert(x, z)

1: if x = null then
2: root[T] ← z; parent[z] ← null;
3: return;
4: if key[x] > key[z] then
5: if left[x] = null then
6: left[x] ← z; parent[z] ← x;
7: else TreeInsert(left[x], z);
8: else
9: if right[x] = null then
10: right[x] ← z; parent[z] ← x;
11: else TreeInsert(right[x], z);
Case 1:
Element does not have any children
  ▶ Simply go to the parent and set the corresponding pointer to null.
Case 1:
Element does not have any children
  ▶ Simply go to the parent and set the corresponding pointer to null.
Case 1:
Element does not have any children
  ▶ Simply go to the parent and set the corresponding pointer to \text{null}.
Case 2:
Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.
Case 2:
Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.
Binary Search Trees: Delete

Case 2:
Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.
Case 3:
Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor
Case 3:
Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor
Case 3:
Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor
Case 3:
Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor
Case 3:
Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor
Case 3:
Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor
Algorithm 9 TreeDelete($z$)

1: if left[$z$] = null or right[$z$] = null
2: then $y \leftarrow z$ else $y \leftarrow \text{TreeSucc}(z)$; select $y$ to splice out
3: if left[$y$] ≠ null
4: then $x \leftarrow \text{left}[y]$ else $x \leftarrow \text{right}[y]$; $x$ is child of $y$ (or null)
5: if $x \neq$ null then parent[$x$] ← parent[$y$]; parent[$x$] is correct
6: if parent[$y$] = null then
7: root[$T$] ← $x$
8: else
9: if $y$ = left[parent[$y$]] then
10: left[parent[$y$]] ← $x$
11: else
12: right[parent[$y$]] ← $x$
13: if $y \neq z$ then copy $y$-data to $z$

select $y$ to splice out
$y$ is child of $x$ (or null)
parent[$x$] is correct
fix pointer to $x$

If $y \neq z$, copy $y$-data to $z$
Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\Theta(h)$, where $h$ denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\Theta(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.
Balanced Binary Search Trees

All operations on a binary search tree can be performed in time \( \Theta(h) \), where \( h \) denotes the height of the tree.

However the height of the tree may become as large as \( \Theta(n) \).

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of \( \Theta(\log n) \).

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.
Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $O(h)$, where $h$ denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $O(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.
Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where $h$ denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.
Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $O(h)$, where $h$ denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $O(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.