18. Binary Search Trees

Trees

Trees are
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

Use
- Decision trees: hierarchic representation of decision rules
- Syntax trees: parsing and traversing of expressions, e.g. in a compiler
- Code trees: representation of a code, e.g. morse alphabet, huffman code
- Search trees: allow efficient searching for an element by value

Examples

Morse alphabet

Start

<table>
<thead>
<tr>
<th>short</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>N</td>
</tr>
<tr>
<td>S</td>
<td>O</td>
</tr>
<tr>
<td>H</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>F</td>
</tr>
<tr>
<td>L</td>
<td>A</td>
</tr>
<tr>
<td>P</td>
<td>I</td>
</tr>
<tr>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>D</td>
<td>O</td>
</tr>
<tr>
<td>O</td>
<td>CH</td>
</tr>
</tbody>
</table>
**Examples**

Expression tree

\[
\frac{3}{5} + 7.0
\]

**Nomenclature**

- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

**Binary Trees**

A binary tree is either
- either a leaf, i.e. an empty tree,
- or an inner leaf with two trees \(T_l\) (left subtree) and \(T_r\) (right subtree) as left and right successor.

In each node \(v\) we store
- a key \(v\).key
- two nodes \(v\).left and \(v\).right to the roots of the left and right subtree.

A leaf is represented by the null-pointer

**Recall: Linked List Node in Java**

```java
class ListNode {
    int key;
    ListNode next;

    ListNode (int key, ListNode next) {
        this.key = key;
        this.next = next;
    }
}
```

ListNode

key (type int)    next (type ListNode)

1 – 5 – 6 – null
**Search Tree and Searching in Java**

```java
public class SearchTree {
    SearchNode root = null; // Wurzelknoten

    // Gibt zurück, ob Knoten mit Schlüssel k existiert
    public boolean contains (int k){
        SearchNode n = root;
        while (n != null && n.key != k){
            if (k < n.key) n = n.left;
            else if (k > n.key) n = n.right;
            else return true;
        }
        return false;
    }

    ... // Einfügen, Löschen
}
```

**Now: tree nodes in Java**

```java
public class SearchNode {
    int key; // Schlüssel
    SearchNode left; // linker Teilbaum
    SearchNode right; // rechter Teilbaum

    // Konstruktor: Knoten ohne Nachfolger
    SearchNode(int k){
        key = k;
        left = right = null;
    }
}
```

**Binary search tree**

A binary search tree is a binary tree that fulfills the search tree property:

- Every node \( v \) stores a key
- Keys in left subtree \( v.left \) are smaller than \( v.key \)
- Keys in right subtree \( v.right \) are greater than \( v.key \)

**Searching**

Input: Binary search tree with root \( r \), key \( k \)
Output: Node \( v \) with \( v.key = k \) or null

\[
v \leftarrow r
\]

while \( v \neq \) null do
    if \( k = v.key \) then
        return \( v \)
    else if \( k < v.key \) then
        \( v \leftarrow v.left \)
    else
        \( v \leftarrow v.right \)
return null
**Height of a tree**

The height \( h(T) \) of a tree \( T \) with root \( r \) is given by

\[
h(r) = \begin{cases} 
0 & \text{if } r = \text{null} \\
1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.}
\end{cases}
\]

The worst case search time is determined by the height of the tree.

**Insertion of a key**

Insertion of the key \( k \):

- Search for \( k \)
- If successful search: output error
- Of no success: insert the key at the leaf reached
- Implementation: devil is in the detail

![Diagram of tree](image)

**Knoten Einfügen in Java**

```java
public boolean add (int k) {
    if (root == null) {root = new SearchNode(k); return true;}
    SearchNode t=root;
    while (k != t.key) {
        if (k < t.key) {
            if (t.left == null){ t.left = new SearchNode(k); return true;}
            else { t = t.left; }
        }
        else { // k > t.key
            if (t.right == null){ t.right = new SearchNode(k); return true;}
            else { t = t.right; }
        }
    }
    return false;
}
```

**Remove node**

Three cases possible:

- Node has no children
- Node has one child
- Node has two children

[Leaves do not count here]
**Remove node**

**Node has no children**

Simple case: replace node by leaf.

![Diagram of a tree with node 4 removed](image1)

\[\text{remove}(4) \rightarrow 8 \ 3 \ 5 \ 13 \ 10 \ 9 \ 19\]

**Remove node**

**Node has one child**

Also simple: replace node by single child.

![Diagram of a tree with node 3 removed](image2)

\[\text{remove}(3) \rightarrow 8 \ 5 \ 4 \ 13 \ 10 \ 9 \ 19\]

**Remove node**

**Node v has two children**

The following observation helps: the smallest key in the right subtree \(v.\text{right}\) (the **symmetric successor** of \(v\))

- is smaller than all keys in \(v.\text{right}\)
- is greater than all keys in \(v.\text{left}\)
- and cannot have a left child.

Solution: replace \(v\) by its symmetric successor.

![Diagram of a tree with node 8 removed](image3)

**By symmetry...**

**Node v has two children**

Also possible: replace \(v\) by its symmetric predecessor.

Implementation: devil is in the detail!

![Diagram of a tree with node 8 removed](image4)

\[\text{remove}(3) \rightarrow 5 \ 10 \ 9 \ 19\]
Algorithm SymmetricSuccessor(v)

Input: Node v of a binary search tree.
Output: Symmetric successor of v

w ← v.right
x ← w.left
while x ≠ null do
  w ← x
  x ← x.left
return w

SymmetricDesc in Java

```java
public SearchNode symmetricDesc(SearchNode node) {
  if (node.left == null) { return node.right; }
  if (node.right == null) { return node.left; }
  SearchNode n = node.right; // cannot be null
  SearchNode parent = null;
  while (n.left != null) { parent = n; n = n.left; }
  if (parent != null) {
    parent.left = n.right;
    n.right = node.right;
  } // else n == node.right
  n.left = node.left;
  return n;
}
```

This algorithm returns the symmetric descendent. But it does even more: it handles also all cases with one or no descendent. And it replaces the symmetric descendent by its successor.

Knoten Löschen in Java

```java
public boolean remove (int k) {
  SearchNode n = root;
  if (n != null && n.key == k) {
    root = SymmetricDesc(root); return true;
  }
  while (n != null) {
    if (n.left != null && k == n.left.key) {
      n.left = SymmetricDesc(n.left); return true;
    } else if (n.right != null && k == n.right.key) {
      n.right = SymmetricDesc(n.right); return true;
    } else if (k < n.key) { n = n.left;
    } else { n = n.right; }
    return false;
  }
```

Traversal possibilities

- preorder: v, then $T_{left}(v)$, then $T_{right}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{left}(v)$, then $T_{right}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{left}(v)$, then v, then $T_{right}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19
Efficiency Considerations

Obviously the runtime of the algorithms search, insert and delete depend in the worst case on the height of the tree.

Degenerated trees are in the worst case thus not better than a linked list.

Balanced trees make sure (e.g. with rotations) during insertion or deletion that the tree stays balanced and provide certain guarantees for the algorithms.

The data structures TreeSet and TreeMap in Java are implemented with balanced trees (so called Red-Black-Trees).

19. Heaps

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap

Binary tree with the following properties:

1. Complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right
3. Heap-Condition:
   Max-(Min-)Heap: key of a child smaller (greater) than that of the parent node
## Heap as Array

**Tree → Array:**
- children(i) = \{2i + 1, 2i + 2\}
- parent(i) = \lfloor (i - 1)/2 \rfloor

![Tree and Array Diagram]

Depends on the starting index\(^\text{10}\)

\(^{10}\)For array that start at 1: \{2i + 1, 2i + 2\} → \{2i, 2i + 1\}, \lfloor (i - 1)/2 \rfloor → \lfloor i/2 \rfloor

## Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively

## Heap in Java

```java
public class Heap {
    double[] A;// will need to grow
    int sz;
    // Heap initialized with 16 elements
    Heap () {
        A = new double[16]; sz = 0;
    }
    // binary growth of the array
    void grow(){ ... }
    // insert element in the heap
    public void add(double value){...}
    // extract and return first (maximal) element
    public double remove() {...}
}
```

## Algorithm Sift-Up(A, m)

**Input:** Array \( A \) with at least \( m + 1 \) and Max-Heap-Structure on \( A[0, \ldots, m - 1] \)

**Output:** Array \( A \) with Max-Heap-Structure on \( A[0, \ldots, m] \).

\( v \leftarrow A[m] \) // value
\( c \leftarrow m \) // current position
\( p \leftarrow \lfloor (c - 1)/2 \rfloor \) // parent node

**while** \( c > 0 \) and \( v > A[p] \) **do**

- \( A[c] \leftarrow A[p] \) // Value parent node → current node
- \( c \leftarrow p \) // parent node → current node
- \( p \leftarrow \lfloor (c - 1)/2 \rfloor \)

\( A[c] \leftarrow v \) // value → current node
add

// insert element to the heap
public void add(double value){
    if (sz == A.length){ grow(); }
    int current = sz;
    int parent = (current−1)/2;
    // sift value up
    while (current > 0 && value > A[parent]) {
        current = parent;
        parent = (current−1)/2;
    }
    A[current] = value;
    sz++;
}

Why this is correct: Recursive heap structure

A heap consists of two heaps:

Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)

Algorithm Sift-down(A, i, m)

Input: Array A with max-heap structure for the children of i. Last element m.
Output: Array A with heap structure for i with last element m.

while 2i + 1 ≤ m do
    j ← 2i + 1; // j left child
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j right child with greater key
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // keep sinking
    else
        i ← m; // sinking finished
**remove**

```java
public double remove() {
    double max = A[0];
    double value = A[sz--];
    int i = 0; int j = 0;
    // sift value down
    while (2 * i + 1 < sz) {
        j = 2 * i + 1; // left child
        if (j < sz - 1 && A[j] < A[j+1]) { ++j; } // right key greater
        if (value < A[j]) { // heap condition still violated
            A[i] = A[j]; i = j; // sift down
        } else { i = sz; } // finished
    }
    A[j] = value;
    return max;
}
```

**Sort heap**

A[1, ..., n] is a Heap.

While n > 1

- swap(A[1], A[n])
- Sink(A, 1, n−1);
- n ← n − 1

```plaintext
7 6 4 5 1 2
swap ⇒ 2 6 4 5 1 7
sink ⇒ 6 5 4 2 1 7
swap ⇒ 1 5 4 2 6 7
sink ⇒ 5 4 2 1 6 7
swap ⇒ 1 4 2 5 6 7
sink ⇒ 4 1 2 5 6 7
swap ⇒ 1 2 4 5 6 7
```

**Height of a Heap**

What is the height $H(n)$ of Heap with $n$ nodes? On the $i$-th level of a binary tree there are at most $2^i$ nodes. Up to the last level of a heap all levels are filled with values.

$$H(n) = \min\{h \in \mathbb{N} : \sum_{i=0}^{h-1} 2^i \geq n\}$$

with $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

thus

$$H(n) = \lceil \log_2(n + 1) \rceil.$$  

**Runtime of the Heap-Algorithms**

The algorithms insert and extract therefore make about $\log_2(n + 1)$ "Steps".11

That makes the heap a very fast data structure because the logarithm grows only very slowly. It is used for sorting data and to implement priority Queues.

---

11 will be made more precise in Computer Science II.