FDS2 - Übung 8

SS 2025

Tim Peko

Inhaltsverzeichnis

24. April 2025

1. Beispiel 1: ADT `bstree`

1.1. Lösungsansatz

Der binäre Suchbaum (`bstree`) als Ganzes verwaltet einen Zeiger auf den Wurzelknoten und einen Zähler für die Anzahl der Knoten im Baum. Jeder Knoten enthält einen Wert vom Typ `int`, sowie Zeiger auf den linken und rechten Kindsknoten.

Die öffentlich angebotenen Methoden lassen sich in folgende Kategorien unterteilen:

1. **Konstruktoren und Destruktor**:
   - `bstree()`: Erzeugt einen leeren Baum
   - `bstree(bstree const& src)`: Kopiert einen bestehenden Baum
   - `~bstree()`: Gibt den Speicher aller Knoten frei
2. **Zugriffsmethoden**:
   - `apply`: Wendet eine Funktion auf jeden Knoten im Baum an (in-order Traversierung)
   - `at`: Gibt den Wert am angegebenen Index zurück (in-order Traversierung)
   - `contains`: Prüft, ob ein Wert im Baum vorhanden ist
   - `count`: Zählt Vorkommen eines bestimmten Wertes im Baum
   - `empty`: Prüft, ob der Baum leer ist
   - `equals`: Vergleicht zwei Bäume auf strukturelle Gleichheit
   - `size`: Gibt die Anzahl der Knoten im Baum zurück
3. **Modifikationsmethoden**:
   - `insert`: Fügt einen Wert in den Baum ein
   - `remove`: Entfernt ein Vorkommen eines Wertes aus dem Baum
   - `remove_all`: Entfernt alle Vorkommen eines Wertes aus dem Baum
   - `clear`: Entfernt alle Knoten aus dem Baum
4. **Ausgabemethoden**:
   - `print`: Eine flache Darstellung des Baums
   - `print_2d`: Eine zweidimensionale Darstellung des Baums von links nach rechts
   - `print_2d_upright`: Eine zweidimensionale Darstellung des Baums von oben nach unten

Für die rekursiven Operationen wurden private Hilfsmethoden implementiert, die die eigentliche Rekursion durchführen. Die öffentlichen Methoden dienen hauptsächlich als Wrapper, die die Gültigkeit der Eingabeparameter prüfen und dann die entsprechenden rekursiven Methoden aufrufen.

Bei der Implementierung der `remove` Methode wurden drei Fälle unterschieden:
1. Löschen eines Leaf-Knotens: Der Knoten wird einfach entfernt
2. Löschen eines Knotens mit einem Kind: Das Kind ersetzt den Knoten
3. Löschen eines Knotens mit zwei Kindern: Der Knoten wird durch den kleinsten Wert im rechten Teilbaum ersetzt

1.1.1. Visualisierungsmethoden

Die drei Ausgabemethoden implementieren unterschiedliche Traversierungs- und Ausgabealgorithmen:

`print()` - In-order Traversierung:
- Rekursive in-order Traversierung (links → Knoten → rechts)
- Wrapper mit eckigen Klammern um Ausgabestring
- Bedingte Ausgabe von < und > basierend auf `node->left/right != nullptr`
- Direkte Ausgabe auf `std::ostream`

`print_2d()` - Depth-First mit Einrückung:
- Rekursive Tiefensuche: rechter Teilbaum → Knoten → linker Teilbaum
- Tiefenparameter für Einrückungsberechnung (`depth * step_size`)
- Bindestriche als Einrückungszeichen ab bestimmter Tiefe
- Zeilenumbruch nach jedem Knoten

`print_2d_upright()` - Level-order mit Platzberechnung:
- Hilfsmethoden: `calculate_space_required_upright()` und `nodes_at_depth()`
- Berechnung des Gesamtplatzbedarfs durch rekursive Maximumsuche
- Level-order Traversierung mit `nullptr`-Platzhaltern für leere Positionen
- Gleichmäßige Platzverteilung: `sub_area_width = space_required / (1 << depth)`
- Zentrierte Knotenpositionierung mit links-/rechts-Padding

1.2. Testfälle

Die Testfälle sind in der Datei `main01.cpp` implementiert und geben die Ergebnisse auf der Konsole aus. Sie wurden in folgende Gruppen unterteilt.

1.2.1. Leerer Baum

```
=== Empty Tree ===
tree = []
tree.size() = 0
tree.empty() returns true -- PASSED
tree.size() returns 0 -- PASSED
tree.contains(42) returns false -- PASSED
tree.count(42) returns 0 -- PASSED
tree.remove(42) returns false -- PASSED
tree.remove_all(42) returns 0 -- PASSED
tree.at(-1, value) returns false -- PASSED
tree.at(0, value) returns false -- PASSED
tree.at(1, value) returns false -- PASSED
tree.clear() returns 0 -- PASSED
tree.empty() returns true -- PASSED
tree.apply(add_one) doesn't crash -- PASSED
tree2 = []
tree2.size() = 0
tree equals tree2 -- PASSED
```

Ergebnis: PASSED

1.2.2. Einzelner Knoten

```
=== Single Node ===
tree = [42]
tree.size() = 1
tree.empty() returns false -- PASSED
tree.size() returns 1 -- PASSED
tree.contains(42) returns true -- PASSED
tree.contains(0) returns false -- PASSED
tree.contains(100) returns false -- PASSED
tree.count(42) returns 1 -- PASSED
tree.count(0) returns 0 -- PASSED
tree.at(0, value) returns true -- PASSED
tree[0] is 42 -- PASSED
tree.at(-1, value) returns false -- PASSED
tree.at(1, value) returns false -- PASSED
tree.remove(0) returns false -- PASSED
tree.size() still 1 after failed remove -- PASSED
tree.remove(42) returns true -- PASSED
tree.size() is 0 after remove -- PASSED
tree.empty() returns true -- PASSED
tree = []
tree.size() = 0
```

Ergebnis: PASSED

1.2.3. Einfügen und Struktur

```
=== Insertion and Tree Structure ===
after inserting 50:
tree.size() = 1
tree.print_2d() =
50

tree.print_2d_upright() =
 50


after inserting 30:
tree.size() = 2
tree.print_2d() =
50
 --30

tree.print_2d_upright() =
   50
 30


after inserting 70:
tree.size() = 3
tree.print_2d() =
 --70
50
 --30

tree.print_2d_upright() =
   50
 30  70


after inserting 20:
tree.size() = 4
tree.print_2d() =
 --70
50
 --30
    --20

tree.print_2d_upright() =
       50
   30      70
 20


after inserting 40:
tree.size() = 5
tree.print_2d() =
 --70
50
    --40
 --30
    --20

tree.print_2d_upright() =
       50
   30      70
 20  40


after inserting 60:
tree.size() = 6
tree.print_2d() =
 --70
    --60
50
    --40
 --30
    --20

tree.print_2d_upright() =
       50
   30      70
 20  40  60
```

```
after inserting 80:
tree.size() = 7
tree.print_2d() =
    --80
 --70
    --60
50
    --40
 --30
    --20

tree.print_2d_upright() =
       50
   30      70
 20  40  60  80


after inserting 10:
tree.size() = 8
tree.print_2d() =
    --80
 --70
    --60
50
    --40
 --30
    --20
       --10

tree.print_2d_upright() =
               50
       30              70
   20      40      60      80
 10

after inserting 25:
tree.size() = 9
tree.print_2d() =
    --80
 --70
    --60
50
    --40
 --30
       --25
    --20
       --10

tree.print_2d_upright() =
               50
       30              70
   20      40      60      80
 10  25

after inserting 35:
tree.size() = 10
tree.print_2d() =
    --80
 --70
    --60
50
    --40
       --35
 --30
       --25
    --20
       --10

tree.print_2d_upright() =
               50
       30              70
   20      40      60      80
 10  25  35
```

```
after inserting 45:
tree.size() = 11
tree.print_2d() =
    --80
 --70
    --60
50
       --45
    --40
       --35
 --30
       --25
    --20
       --10

tree.print_2d_upright() =
              50
       30             70
   20      40      60      80
 10  25  35  45


tree = [10 < 20 > 25 < 30 > 35 < 40 > 45 < 50 > 60 < 70 > 80]
tree.size() = 11
size() is correct -- PASSED
all inserted values are contained -- PASSED

testing at() method (in-order access):
tree[0] = 10
tree[1] = 20
tree[2] = 25
tree[3] = 30
tree[4] = 35
tree[5] = 40
tree[6] = 45
tree[7] = 50
tree[8] = 60
tree[9] = 70
tree[10] = 80

testing duplicates:
tree = [10 < 20 > 25 < 30 > 30 < 35 < 40 > 45 < 50 > 50 < 60 < 70 > 80]
tree.size() = 13
size increased after duplicates -- PASSED
count(50) returns 2 -- PASSED
count(30) returns 2 -- PASSED
```

Ergebnis: PASSED

1.2.4. Copy Constructor

```
═══ Copy Constructor ═══
original = [3 < 5 > 7 < 10 > 15]
original.size() = 5
copy = [3 < 5 > 7 < 10 > 15]
copy.size() = 5
copy.size() == original.size() -- PASSED
copy equals original -- PASSED
original after adding 20 = [3 < 5 > 7 < 10 > 15 > 20]
original after adding 20.size() = 6
copy after original modification = [3 < 5 > 7 < 10 > 15]
copy after original modification.size() = 5
copy.equals(original) returns false -- PASSED
copy.size() == 5 -- PASSED
original.size() == 6 -- PASSED
empty_copy.empty() returns true -- PASSED
empty_tree.equals(empty_copy) returns true -- PASSED
```

Ergebnis: PASSED

1.2.5. Entfernen Edge Cases

```
═══ Removal Edge Cases ═══
tree1 = [5 < 10 > 15]
tree1.size() = 3
tree1.remove(5) returns true -- PASSED
tree1 = [10 > 15]
tree1.size() = 2
tree1.size() == 2 -- PASSED
tree1.contains(5) returns false -- PASSED
tree2 = [3 < 5 < 10]
tree2.size() = 3
tree2.remove(5) returns true -- PASSED
tree2 = [3 < 10]
tree2.size() = 2
tree2.contains(3) returns true -- PASSED
tree3 = [3 < 5 > 7 < 10 > 12 < 15 > 20]
tree3.size() = 7
tree3.remove(15) returns true -- PASSED
tree3 = [3 < 5 > 7 < 10 > 12 < 20]
tree3.size() = 6
tree3.contains(15) returns false -- PASSED
all other nodes still contained -- PASSED
tree4 = [5 < 10 > 15]
tree4.size() = 3
tree4.remove(10) returns true -- PASSED
tree4 = [5 < 15]
tree4.size() = 2
tree4.contains(10) returns false -- PASSED
other nodes still contained -- PASSED
after removing 5:
tree5.size() = 2
 --15
10

size decreased -- PASSED
after removing 10:
tree5.size() = 1
15

size decreased -- PASSED
after removing 15:
tree5.size() = 0
nulltree
size decreased -- PASSED
tree5 is empty after removing all -- PASSED
```

Ergebnis: PASSED

1.2.6. Entfernen von Werten

```
=== Remove All ===
tree = [5 > 5 < 10 > 10 > 10 > 10 < 15]
tree.size() = 7
tree.count(10) = 4
tree.count(5) = 2
tree.remove_all(10) returned 4 -- PASSED
tree.contains(10) returns false -- PASSED
tree.count(10) is 0 -- PASSED
tree.contains(5) returns true -- PASSED
tree = [15]
tree.size() = 1
tree.remove_all(5) returned 2 -- PASSED
tree.contains(5) returns false -- PASSED
tree.remove_all(100) returned 0 -- PASSED
tree = [15 > 42]
tree.size() = 2
tree.remove_all(42) returned 1 -- PASSED
tree.contains(42) returns false -- PASSED
```

Ergebnis: PASSED

1.2.7. Funktion anwenden

```
=== Apply Function ===
tree = [3 < 5 > 7 < 10 > 15]
tree.size() = 5
after applying add_one (x ⇒ x + 1):
tree = [4 < 6 > 8 < 11 > 16]
tree.size() = 5
tree.contains(11) returns true -- PASSED
tree.contains(6) returns true -- PASSED
tree.contains(16) returns true -- PASSED
tree.contains(4) returns true -- PASSED
tree.contains(8) returns true -- PASSED
tree.contains(10) returns false -- PASSED
tree.contains(5) returns false -- PASSED
after applying multiply_by_two (x ⇒ x * 2):
tree = [8 < 12 > 16 < 22 > 32]
tree.size() = 5
apply on empty tree doesn't crash -- PASSED
```

Ergebnis: PASSED

1.2.8. Ausführliches Indexing

```
=== Comprehensive at() Method ===
tree = [20 < 30 > 40 < 50 > 60 < 70 > 80]
tree.size() = 7

testing all valid indices:
tree[0] = 20 (success: 1)
tree.at(0) succeeds -- PASSED
tree[1] = 30 (success: 1)
tree.at(1) succeeds -- PASSED
tree[2] = 40 (success: 1)
tree.at(2) succeeds -- PASSED
tree[3] = 50 (success: 1)
tree.at(3) succeeds -- PASSED
tree[4] = 60 (success: 1)
tree.at(4) succeeds -- PASSED
tree[5] = 70 (success: 1)
tree.at(5) succeeds -- PASSED
tree[6] = 80 (success: 1)
tree.at(6) succeeds -- PASSED
tree.at(-1) fails -- PASSED
tree.at(-100) fails -- PASSED
tree.at(size) fails -- PASSED
tree.at(size+1) fails -- PASSED
tree.at(1000) fails -- PASSED
```

Ergebnis: PASSED

1.2.9. Baum leeren

```
=== Clear Method ===
tree = [3 < 5 > 7 < 10 > 15]
tree.size() = 5
tree_after_clear = []
tree_after_clear.size() = 0
tree.clear() returned original size -- PASSED
tree is empty after clear -- PASSED
tree.size() is 0 after clear -- PASSED
tree.contains(10) returns false after clear -- PASSED
tree.remove(10) returns false after clear -- PASSED
second clear() returns 0 -- PASSED
tree is still empty after second clear -- PASSED
```

Ergebnis: PASSED

1.2.10. Bäume vergleichen

```
═══ Equals Method ═══
tree1 = [3 < 5 > 7 < 10 > 15]
tree1.size() = 5
tree2 = [3 < 5 > 7 < 10 > 15]
tree2.size() = 5
identical trees are equal -- PASSED
equals is symmetric (t2 == t1 is t1 == t2) -- PASSED
tree2 = [3 < 5 > 7 < 10 > 15 > 20]
tree2.size() = 6
trees with different sizes are not equal -- PASSED
tree3 = [3 < 5 > 7 < 10 < 15]
tree3.size() = 5
same values different structure are not equal -- PASSED
tree4 = [10 > 10]
tree4.size() = 2
tree5 = [10 > 10]
tree5.size() = 2
trees with same duplicates are equal -- PASSED
tree5 = [10 > 10 > 10]
tree5.size() = 3
trees with different duplicate counts are not equal -- PASSED
empty trees are equal -- PASSED
empty tree not equal to non-empty -- PASSED
```

Ergebnis: PASSED

1.2.11. Edge Cases

```
═══ Extreme Cases ═══
neg_tree = [-20 > -15 < -10 > -5 > -1]
neg_tree.size() = 5
negative numbers work correctly -- PASSED
zero_tree = [-1 < 0 > 1]
zero_tree.size() = 3
zero works correctly -- PASSED
large_tree = [999999 < 1000000 > 1000001]
large_tree.size() = 3
large numbers work correctly -- PASSED
dup_tree = [42 > 42 > 42 > 42 > 42 > 42 > 42 > 42 > 42 > 42]
dup_tree.size() = 10
many duplicates work correctly -- PASSED
degen_tree = [1 > 2 > 3 > 4 > 5 > 6 > 7 > 8 > 9 > 10]
degen_tree.size() = 10
degenerate tree works correctly -- PASSED
rev_degen_tree = [1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10]
rev_degen_tree.size() = 10
reverse degenerate tree works correctly -- PASSED
```

Ergebnis: PASSED

Aufwand in h: 5