DATA STRUCTURES AND ALGORITHMS

Lecture 11

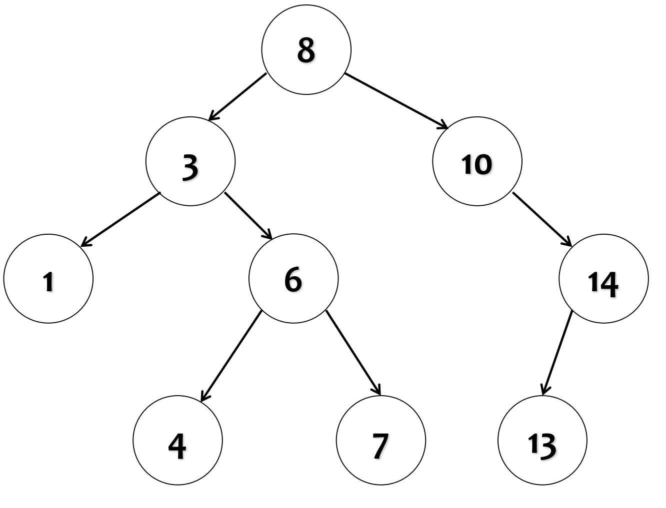
Learning outcome 4

BINARY SEARCH TREES

Introduction

- Binary search tree (BST) is a subtype of binary tree with the following properties:
  - All data in the left subtree is smaller than the data in the root of the subtree
  - All data in the right subtree is greater than or equal to the data in the root of the subtree
  - Each subtree is also a binary search tree
- The main advantage of BST is the ability to efficiently search the tree to find some value

BST example



Searching BST

- BST search processes one node at each level, which allows us to achieve logarithmic complexity (depends on the shape of the tree)
  - Let's say we're looking for value 7
  - We start from the root and according to its value (8) we know that the value 7 is certainly in the left subtree (because 7 < 8)
  - We look at the root of the left subtree (3) and know that the value of 7 is certainly in the right subtree (because 7 > 3)
  - We look at the root of the right subtree (6) and know that the value 7 is certainly in the right subtree (because 7 > 6)
  - We look at the root of the right subtree and we've found 7

Inserting into BST

- Insertion can also be very efficient:
  - Let's say we want to insert a value of 4
  - We start from the root and according to its value (8) we know that the value 4 should be placed in the left subtree (because 4 < 8)
  - We look at the root of the left subtree (3) and we know that the value 4 should be put in the right subtree (because 4 > 3)
  - We look at the root of the right subtree (6) and we know that the value 4 should be put in the left subtree (because 4 < 6)
  - We look at the root of the left subtree (4) and we know that the value 4 should be put in the right subtree (because 4 = 4)
    - The right subtree does not exist, so we create a new node of value 4 and place it as the right child of the existing node of value 4

AVL TREES

Introduction

- The BST can deviate significantly from the complete tree
- Lets go to www.cs.usfca.edu/~galles/visualization/BST.html
  - Add values: 5, 4, 6, 3, 4, 5, 10
    - Perfect tree, optimal search
  - Reset the tree and add same values, just in different order: 3, 4, 4, 5, 5, 6, 10
    - We get a diagonal tree whose performance is equal to the performance of the list

AVL trees

- AVL trees are a subtype of the binary search tree with the following properties:
  - Both subtrees of the node are either of equal depth or the difference in depth is equal to 1
    - This means that each leaf node is approximately equally away from the root
  - When inserting (or deleting) a node, you may need to balance the tree with one or more rotations to keep the tree balanced
- The AVL tree is named after its creators: G. M. Adelson-Velskii and E. M. Landis
  - It was the first balancing tree

Searches and insertion in AVL trees

- The search method is the same as that of BST
  - Since the tree is balanced, the search time is always $O(\log n)$, which makes it great for searching
  - Insert / delete performance suffers because of rotations
- The insertion of the node is done in two phases:
  - The insertion is done in the same way as with BST
  - For all ancestors of the inserted node, a balance factor is calculated that is equal to: depth of the left subtree minus depth of the right subtree
    - If the factor is -1, 0 or +1 the tree is balanced
    - If the balancing factor is -2 or +2, rotation balancing is performed

DEMO

- www.cs.usfca.edu/~galles/visualization/AVLtree.html
- Create AVL tree with values 1 to 15

RED BLACK TREES

Introduction

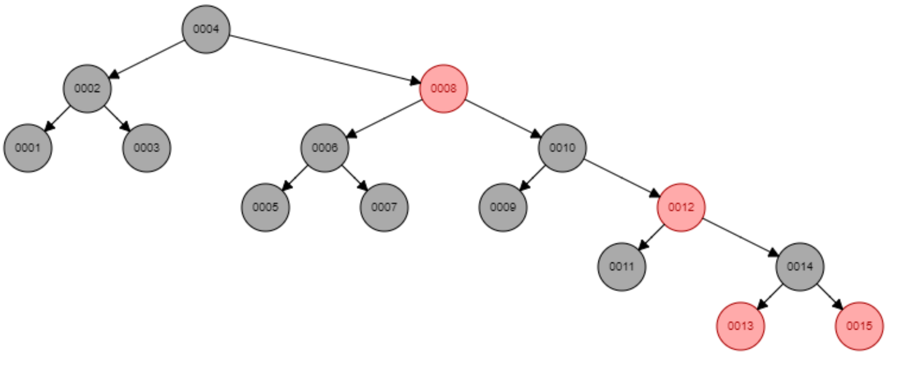
- Red-black trees (RB trees) are also a subtype of BST with the following properties:
  - They are balancing
  - Each node contains an additional bit of information containing the color (red / black) used when inserting / deleting to keep the tree roughly balanced
    - The root is always black
    - If a node is red, both children must be black
    - Each path from a node to a leaf must contain an equal number of black nodes
    - It follows from the above that there must never be two consecutive red nodes on a path (but there may be many black ones)

Inserting a node

- Insertion begins as with BST, with:
  - The new node is always red
  - When we place it in position, there is a chance that the RB tree property is lost
  - Rotation and painting restore the property of the RB tree

DEMO

- www.cs.usfca.edu/~galles/visualization/RedBlack.html
- Create a RB tree with values 1 to 15



AVL tree vs RB tree

- These are the two most famous self-balancing trees
  - Both trees use rotations to keep the tree (roughly) balanced after inserting / changing nodes
- Search is generally faster in AVL trees because all nodes are either of equal depth or the difference is in one level
  - RB tree deviates a bit more, but it also has a search in $O(\log n)$
- Both trees guarantee $O(\log n)$ for insertion / deletion
  - AVL guarantees additional rotations in $O(\log n)$
  - RB additional rotations are guaranteed in $O(1)$
  - => Insertion / deletion is generally faster in RB trees
- C++, Java, C# … use RB trees

DICTIONARIES

Introduction

- Dictionary (associative array, map, symbol table) is a container that contains a collection of pairs (key, value) and which provides operations:
  - Adding a new pair
  - Removing a pair by the key
  - Modifying the existing value (but not the key)
  - Retrieving value by the key (emphasis is on this!)
- Built-in types in some programming languages (Python)
- Two main options for dictionary implementation are:
  - Hash tables (LO6)
  - BST subtypes (LO4, this one)

Comparison of dictionary implementation options

Underlying data structure	Lookup		Insertion		Deletion		Ordered
	average	worst case	average	worst case	average	worst case	
Hash table	O(1)	O(*n*)	O(1)	O(*n*)	O(1)	O(*n*)	No
Self-balancing binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)	Yes
unbalanced binary search tree	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	O(log *n*)	O(*n*)	Yes
Sequential container of key-value pairs (e.g. association list)	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)	O(*n*)	No

Taken from: en.wikipedia.org

Dictionaries using BST

- STL contains four types of dictionaries implemented using BSTs: `set`, `multiset`, `map` and `multimap`
  - RB trees are used in the implementation (implementer's decision)
- `map` keeps values stored under unique keys
- `multimap` allows values with duplicate keys
- `set` keeps only unique keys (i.e. value = key)
- `multiset` allows duplicate keys
- In all structures the elements are sorted
  - When we traverse the RB tree with the INORDER algorithm

Dictionary examples

- List of all products offered by the store stored under a unique product code
  - `map<string, Product>`
- List of all invoices issued to an OIB
  - `multimap<string, Invoice>`
- List of lottery numbers from last night’s draw
  - `set<int>`
- List of all license plates that have parked in the garage since the opening of a new campus
  - `multiset<string>`

SET AND MULTISET

Creating and destroying sets/multisets (1/2)

- There are four basic ways to create a set/multiset:
  - `set<int> one;`
    - Creates an empty set
  - `set<int> two(one.begin(), one.end());`
    - Creates a set of all elements within a range [begin, end)
  - `set<int> three(dva);`
    - Creates a set by copying all elements from another set
  - `set<int> four({ 11, 22, 33, 22, 44 });`
    - Creates a set based on the initialization list (by copying each of the values)
    - If we give a set two equal values, the set will simply ignore the other
  - For multiset, just add „multi" prefix

Creating and destroying sets/multisets (2/2)

- The set / multiset is automatically destroyed when the function ends
  - If it stores objects, a destructor is called on each
- `operator=` copies the contents of one set/multiset to another
  - The previous content of the second set/multiset is destroyed
- The values put in set/multiset cannot be changed

set/multiset iterators

- The most important iterators are:
  - `set<T>::iterator` is a class whose ++ moves towards the end
  - `set<T>::reverse_iterator` is a class whose ++ moves towards the beginning
- Since the sets are sorted:
  - By using the iterator in one direction, we go from smaller to larger values
  - By using the iterator in the other direction, we go from higher to lower values
- If we store objects in a set, overloading the `operator<` defines which object is smaller and which is larger

Structure `pair<T1,T2>`

- Structure `pair<T1,T2>` represents a pair of values
  - First one is named `first` and has a type `T1`
  - Second one is named `second` and has a type `T2`
- Example:

```
pair<int, string> p(17, "Miro Miric");
cout << p.first << " " << p.second << endl;
p.first++;
cout << p.first << " " << p.second << endl;
```

Adding to a set

- We can insert values into a set in three main ways:
  - `s.insert(x)` copies x and places it in its position in the set
    - Returns an object of type `pair<iterator, bool>`
      - `first` points to either a freshly inserted element or an element that already exists in the set
      - `second` contains `true` (if the insertion was successful) or `false` (if the value already existed in the set)
  - `s.insert(begin, end)` copies elements [begin, end) and places them in their position in the set
    - Returns void
  - `s.insert({ 11, 22, 33 })` copies the numbers and places them in their positions in the set
    - Returns void

Removing from a set

- We can delete values from a set in four main ways:
  - `s.erase(val)` deletes an element equal to the val
    - Returns the number of deleted items (0 or 1)
  - `s.erase(position)` deletes whichever element is in the said position
    - Returns the iterator to the element behind the deleted element
  - `s.erase(begin, end)` deletes elements in the specified range [begin, end)
    - Returns the iterator to the element immediately behind the last deleted element
  - `s.clear()` removes and destroys all elements of the set

Example

```
set<int> s({ 55, 11, 55, 33, 22, 44 });
cout << s.size() << endl;

auto ir = s.insert(11);
cout << "Inserted: " << ir.second << endl;
ir = s.insert(66);
cout << "Inserted: " << ir.second << endl;

s.erase(s.begin());
s.erase(66);

for (auto it = s.begin(); it != s.end(); ++it) {
    cout << *it << endl;
}
```


Multiset differences in insert and delete

- The multiset behaves the same as the set, with differences:
  - `s.insert(x)` copies x and places it in his position in the multiset
    - Always succeeds
    - Returns the iterator to the inserted element
  - `s.erase(val)` returns the number of deleted items (0, 1, 2, …)

Other important methods of the set

- `s.find(x)` searches for element x in the set and returns its position
  - If not found, it returns `s.end()`
- `s.count(x)` returns the number of occurrences of element x in the set
  - It can return 0 or 1 because the values are unique
- `s.size()` returns the number of elements in the set
- `s.empty()` returns if the set is empty

ALGEBRA

Multiset differences

- `ms.count(x)` returns the number of occurrences of element x in the multiset
  - There may be 0 or more
- `ms.find(x)` searches for the first element x in the multiset and returns the iterator to its position
  - If not found, it returns `s.end()`
- If we want to retrieve all occurrences of x in the multiset:
  - `ms.equal_range(x)`
    - Returns `pair<iterator, iterator>`
      - `first` is an iterator to a first position
      - `second` is an iterator to a last + 1 position

ALGEBRA

Example

```
multiset<int> ms({ 22, 11, 55, 22, 33, 22, 44 });

auto it = ms.find(22);
cout << *it << endl;

auto range = ms.equal_range(22);
for (auto it = range.first; it != range.second; ++it) {
    cout << *it << endl;
}
```


MAP AND MULTIMAP

Introduction

- Folder and multimap can be understood as a set where key and value are different
  - The keys must be unique in the map, but not in the multimap
  - Pairs are sorted by keys
  - Keys are immutable, values can change

Map and multimap specifics (1/2)

- The interface is very similar, with a few differences:
  - Set/multiset holds keys, while map/multimap holds pairs
    - `first` contains a key, `second` contains a value
  - Iterator points to a pair
  - Parameter for `insert` is a pair
    - For example:

```
map<char, string> m;
m.insert({ 'c', "Canada" });
m.insert(pair<char, string>('a', "America"));
m.insert(pair<char, string>('j', "Japan"));

for (auto it = m.begin(); it != m.end(); ++it) {
   cout << it->first << " " << it->second << endl;
}
```

Map and multimap specifics (2/2)

- `s[key]` retrieves the value stored under the key or inserts a new empty value if the key does not exist
  - Exists only on the map
- `s.at(key)` does the same thing, but throws an exception if the key does not exist
  - Exists only on the map

```
map<char, string> m;
m.insert(pair<char, string>('c', "Canada"));
m.insert(pair<char, string>('a', "America"));

cout << m['c'] << endl;
cout << m['a'] << endl;
cout << m['r'] << endl;
cout << m.size() << endl;
cout << m.at('c') << endl;
cout << m.at('f') << endl;
```


Problem

- Insert numbers from 1 to 100,000 into the vector and into the set. Display how long it takes to search for the number 100,000 in a vector and how long in a set.
  - The search in the vector lasts: 172567 microseconds
    - Must perform 100.000 operations
  - The search in the set lasts: 426 microseconds
    - Must perform log(100.000) = 17 operations

Solution (1/2)

```
// Preparing.
int n = 100000;
vector<int> v(n);

for (int i = 1; i <= n; i++) {
    v.push_back(i);
}

set<int> s(v.begin(), v.end());

// Executing.
auto begin = chrono::high_resolution_clock::now();
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it == n) {
        cout << "Found in vector" << endl;
        break;
    }
}
```


Solution (2/2)

```
auto end = chrono::high_resolution_clock::now();
cout
    << "Vector: "
    << chrono::duration_cast<chrono::microseconds>(end -
begin).count() << " us" << endl;

begin = chrono::high_resolution_clock::now();
if (s.find(n) != s.end()) {
    cout << "Found in set" << endl;
}
end = chrono::high_resolution_clock::now();
cout
    << "Set: "
    << chrono::duration_cast<chrono::microseconds>(end -
begin).count() << " us" << endl;
```