Binary search trees

1. Background
2. Procedures
3. STL containers

1. Background

Terminology related to binary trees:

- left child, right child
- left subtree, right subtree
- leaf
- height

In a binary search tree, each node *x* has a key: *x.key* (common interpretation: *x.key* is an integer)

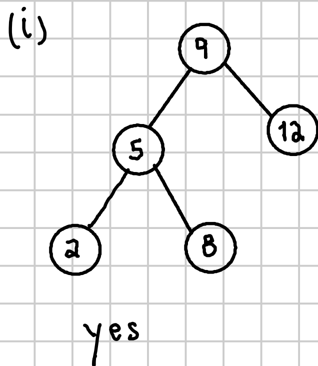
Q: What is a binary search tree (BST)?

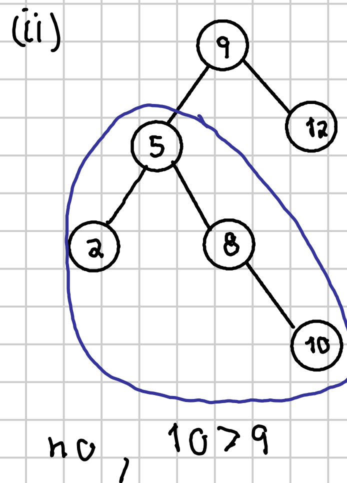
A: A binary tree is a binary search tree, when each node x has the following property:

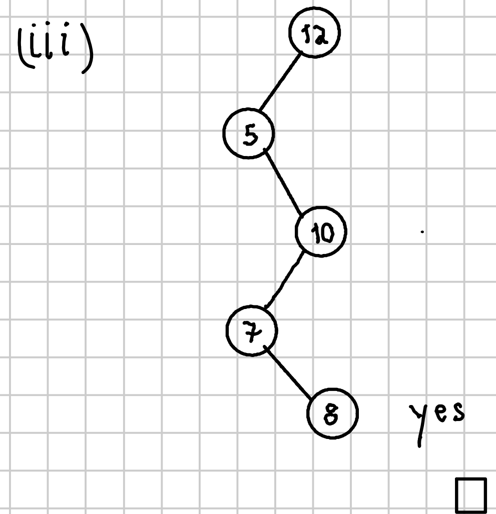
> When y is any node in the left subtree of x, then $y.key < x.key$. When z is any node in the right subtree of x, then $z.key > x.key$.

Example

Is the tree a BST or not?







□

Q: For searching purposes, what makes a BST efficient or inefficent?

A: A BST is efficient when it is balanced.

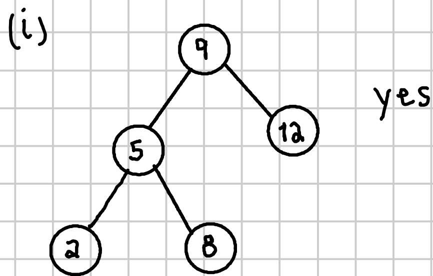
Q: What is a height-balanced BST?

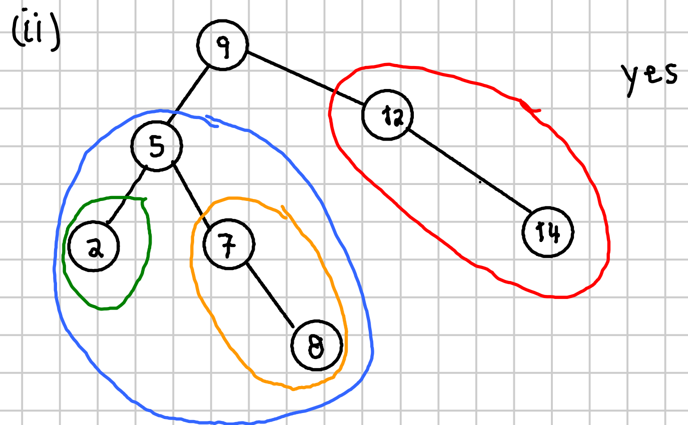
A: In a height-balanced BST, each node x has the following property:

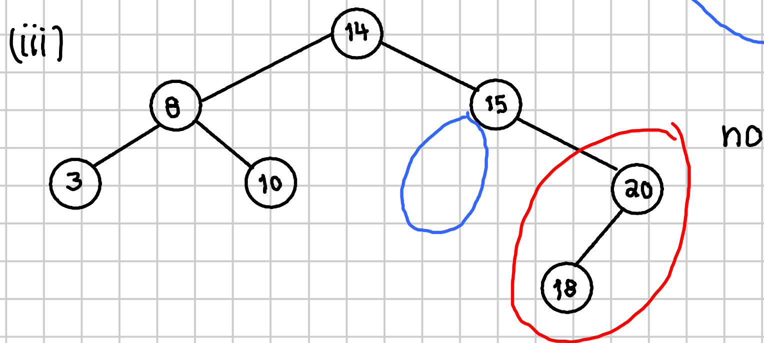
The difference between the heights of the two subtrees of x is at most 1.

Example

Is the BST height-balanced or not?







□

Other kinds of balancing:

For each node x, the difference between the number of nodes in its left and right subtrees of x is at most 1.

2. Procedures

There are numerous operations that can be done on a BST:

- **search**: search for a given *key* in an existing BST
- **insert**: insert a new node with a given *key* into an existing BST
- **delete**: remove a new node having a given *key* from an existing BST, if such a node exists
- **min**: obtain the node with the smallest key from an existing BST
- **max**: obtain the node with the largest key from an existing BST
- **predecessor**: given a node x, obtain the node having the largest key that is less than $x.key$
- **successor**: given a node x, obtain the node having the smallest key that is greater than $x.key$
- **inorder-traversal**: perform a traversal of all the nodes in a BST in ascending order of the keys

For pseudocode we will use the following for binary tree:

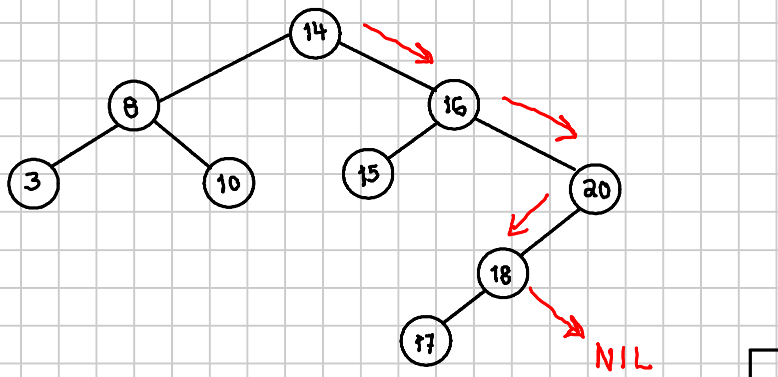
- $Node.key$ = the key of a node
- $Node.parent$ = pointer to parent of Node or NIL if root
- $Node.left$ = pointer left child for binary tree or NIL if child does not exist
- $Node.right$ = pointer right child for binary tree or NIL if child does not exist

Search

```
1   B–TREE–SEARCH(x, k)
2   given BST whose root is x, returns the node whose key is k, when
3   such a node exists, otherwise returns NIL
4   while x ≠ NIL and x.key ≠ k do
5      if k < x.key then
6         x = x.left
7      else
8         x = x.right
9      end
10  end
11  return x
```

Example

B-TREE-SEARCH((14), 19)



□

Min

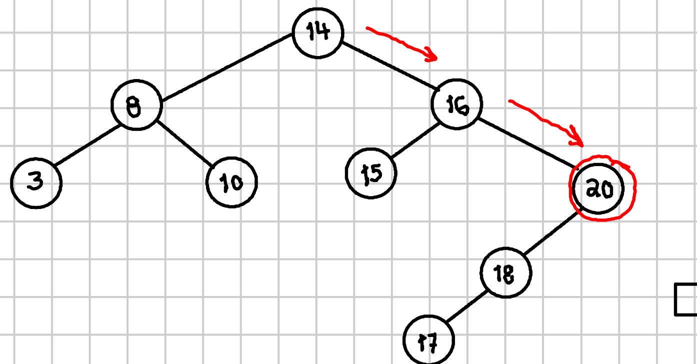
```
1   B–TREE–MIN(x)
2   given BST whose root is x, returns the node having smallest key
3   while x.left ≠ NIL do
4      x = x.left
5   end
6   return x
```

Max

```
1  B–TREE–MAX(x)
2  given BST whose root is x, returns the node having largest key
3  while x.right ≠ NIL do
4     x = x.right
5  end
6  return x
```

Example

B-TREE-MAX(14)



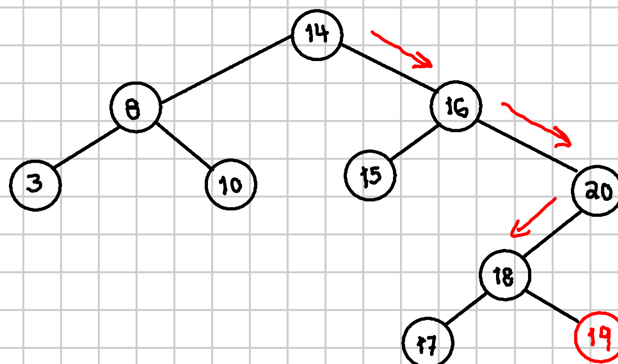
□

Insert

```
1   B-TREE-INSERT(root, x)
2   insert into given BST new node x
3   parent = NIL
4   node = root
5   /* find the correct location for the new node */
6   while node ≠ NIL do
7      parent = node
8      if x.key < node.key then
9         node = node.left
10     else
11        node = node.right
12     end
13  end
14  x.parent = parent
15  /* If the new node has no parent, it is the root. */
16  if x.parent == NIL then
17     root = x
18  else
19
20     /* Is the new node is a left child or a right child? */
21     if x.key < parent.key then
22        parent.left = x
23     else
24        parent.right = x
25     end
26  end
27  x.left = NIL, x.right = NIL
```

Example

INSERT(14 , 19)



Successor

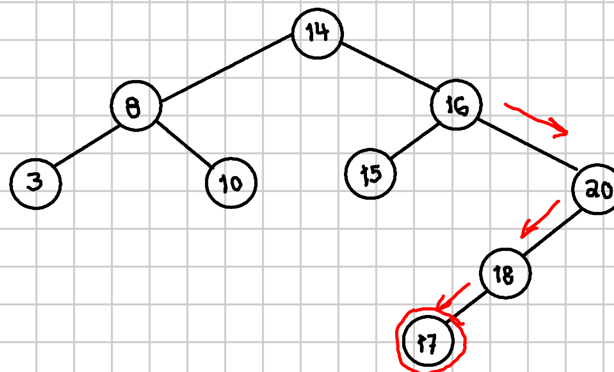
```
1   B–TREE–SUCCESSOR(x)
2   given a node x in a BST, locates the node whose key is the
3   smallest that is greater than x.key
4   if x.right ≠ NIL then
5     return B–TREE–MIN(x.right)
6   end
7   parent = x.parent
8   while parent ≠ NIL and x ≠ parent.left do
9     x = parent, parent = x.parent
10  end
11  return parent
```

Predecessor

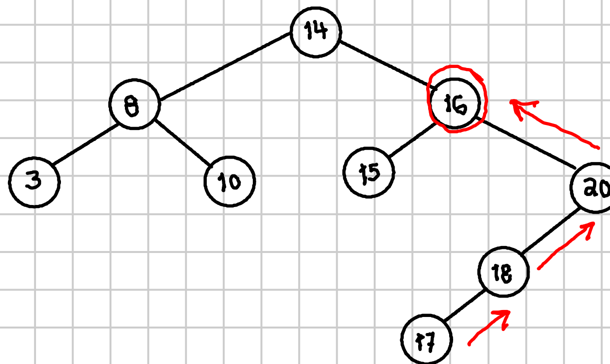
```
1   B–TREE–PREDECESSOR(x)
2   given a node x in a BST, locates the node whose key is the
3   largest that is less than x.key
4   if x.left ≠ NIL then
5     return B–TREE–MAX(x.left)
6   end
7   parent = x.parent
8   while parent ≠ NIL and x ≠ parent.right do
9     x = parent, parent = x.parent
10  end
11  return parent
```

Example

SUCCESSOR(16)



PREDECCESSOR(17)

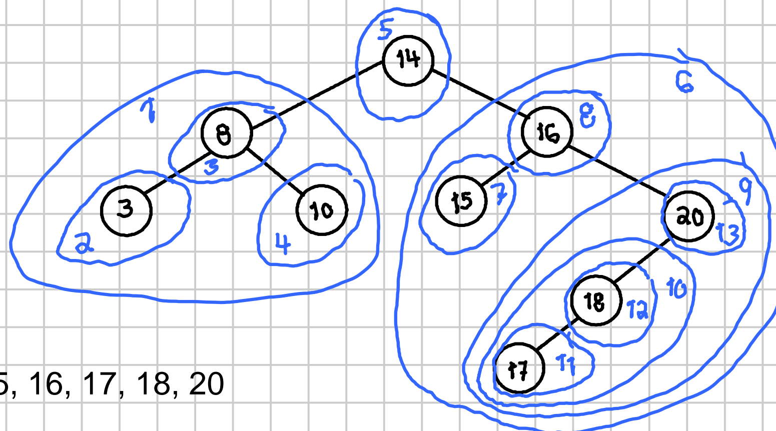


Inorder traversal

```
1   INTRAVERSAL(node)
2   input node tree node
3   if node ≠ NIL then
4     INTRAVERSAL(node.left)
5     ▷ perform some operation using node
6     INTRAVERSAL(node.right)
7   end
```

order of parts: left subtree, root, right subtree

Example



output list: 3, 8, 10, 14, 15, 16, 17, 18, 20

The **delete** procedure is tricky and ill not be presented.

3. STL containers

In C++ `map` and `set` are balanced binary search trees.

From https://en.cppreference.com/w/cpp/container/map

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```
(1)

```
namespace pmr {
    template <class Key, class T, class Compare = std::less<Key>>
    using map = std::map<Key, T, Compare,
                         std::pmr::polymorphic_allocator<std::pair<const Key,T>>>
}
```
(2) (since C++17)

`std::map` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function `Compare`. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees.

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements or nodes (since C++17) (public member function)
`insert_or_assign` (C++17)	inserts an element or assigns to the current element if the key already exists (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`emplace_hint` (C++11)	constructs elements in-place using a hint (public member function)
`try_emplace` (C++17)	inserts in-place if the key does not exist, does nothing if the key exists (public member function)
`erase`	erases elements (public member function)
`swap`	swaps the contents (public member function)
`extract` (C++17)	extracts nodes from the container (public member function)
`merge` (C++17)	splices nodes from another container (public member function)

Lookup

`count`	returns the number of elements matching specific key (public member function)
`find`	finds element with specific key (public member function)
`contains` (C++20)	checks if the container contains element with specific key (public member function)
`equal_range`	returns range of elements matching a specific key (public member function)
`lower_bound`	returns an iterator to the first element *not less* than the given key (public member function)
`upper_bound`	returns an iterator to the first element *greater* than the given key (public member function)

Tämä teos on lisensoitu Creative Commons Nimeä-EiKaupallinen-EiMuutoksia 4.0 Kansainvälinen -lisenssillä. Tarkastele lisenssiä osoitteessa http://creativecommons.org/licenses/by-nc-nd/4.0/.

tekijä: Frank Cameron

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

made by Frank Cameron