C++ Data Structures

Code Snippets Documentation

Competitive Programming Reference

2025-11-14

Contents

Binary Search Tree

A binary search tree is a binary tree data structure that supports searching, insertion and removal of elements, by using the following rules:

1. The left subtree of a node contains only nodes with keys less than or equal to the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. The left and right subtree each must also be a binary search tree.

Trigger

`bst` | `binsearchtree`

Classes

The snippet creates `BSTAbstract`, `BST` classes and `Node` structure.

- `BSTAbstract` class template gets `T` type as a template parameter and `Comp` as "is `T a` less than `T b`" comparison function
- `BST` class is a wrapper for `BSTAbstract` with `Comp` defined as `[](T a, T b) { return a < b; }`

Methods

Initialization

The tree accepts a comparison function `cmp(T a, T b)` when created. It works like a standard `less` comparator.

Default comparator:

```cpp
bool cmp(T a, T b) { return a < b; }
```

The comparator determines the ordering of elements and where new nodes are inserted.

`void insert(T val)`

Inserts a new value into the binary search tree. The function recursively descends left or right according to the comparison rule and places the new value at the correct leaf position.

`T* search(T val)`

Looks for the given value in the tree.

Returns:

- Pointer to the stored value if found
- `nullptr` if the value is not present

`bool remove(T val)`

Removes a value from the tree.

Returns:

- `true` if the value was found and deleted
- `false` if the tree does not contain the value

`void print()`

Prints the BST in a rotated tree-like format:

```
        8
    7
        6
5
```

```
        4
    3
        2
```

Right subtree is displayed above the root, left subtree below it.

Example

```cpp
BST<int> tree;

tree.insert(5);
tree.insert(3);
tree.insert(7);

int* p1 = tree.search(3);  // Found
int* p2 = tree.search(10); // nullptr

tree.remove(3);

tree.print();
```

AVL Tree

This is the same data structure as binary search tree, but with additional balancing and 4 types of rotations to keep the tree balanced (every operation has $O(\log n)$ complexity in worst case).

Supports the same methods as binary search tree, but also supports autobalancing by rotations and counting heights of subtrees.

Trigger

`avltree` | `avl`

Classes

The snippet creates `AVLAbstract` and `AVL` classes, with the same logic as for binary search tree.

- `AVLAbstract` class template with type parameter `T` and comparison function `Comp`
- `AVL` class wrapper with default comparator

Methods

All methods from Binary Search Tree are supported with the same interface. Additionally, the tree automatically maintains balance through rotations after insertions and deletions.

Example

```cpp
AVLTree<int> tree;

tree.insert(1);
tree.insert(2);
tree.insert(3);

int* p1 = tree.search(3);  // Found
int* p2 = tree.search(4);  // nullptr

tree.remove(2);
tree.print();
```

Heap

A binary heap is a complete binary tree data structure that satisfies the heap property. It is implemented using an array representation where for any node at index i:

- Left child is at index $2i + 1$
- Right child is at index $2i + 2$
- Parent is at index $\frac{i-1}{2}$

The heap supports efficient insertion and removal of the root element (min or max depending on comparator).

Trigger

`heap` | `minheap` | `maxheap`

Classes

The snippet creates `Heap` class template and predefined type aliases.

- `Heap` class template gets `T` type and `Comp` as comparison function template parameters
- Type aliases `MinHeap` and `MaxHeap` are wrappers for `Heap` with predefined comparators

Predefined comparators:

```cpp
template <typename T> bool minCompare(const T &a, const T &b) { return a < b; }
template <typename T> bool maxCompare(const T &a, const T &b) { return a > b; }
```

Type aliases:

```cpp
template <typename T = int> using MinHeap = Heap<T, minCompare<T>>;
template <typename T = int> using MaxHeap = Heap<T, maxCompare<T>>;
```

Methods

Initialization

The heap accepts a comparison function `Comp(const T& a, const T& b)` as a template parameter.

- `MinHeap` maintains smallest element at root
- `MaxHeap` maintains largest element at root

`void insert(T val)`

Inserts a new value into the heap. The element is added at the end of the array and then sifted up to maintain the heap property.

`T popRoot()`

Removes and returns the root element (minimum for MinHeap, maximum for MaxHeap). The last element replaces the root, then sifts down to restore the heap property.

Throws:

- `std::out_of_range` if heap is empty

`T peek() const`

Returns the root element without removing it.

Throws:

- `std::out_of_range` if heap is empty

`bool empty() const`
Checks if the heap is empty.

Returns:

- `true` if heap contains no elements
- `false` otherwise

`int size() const`
Returns the number of elements in the heap.

`void clear()`
Removes all elements from the heap.

`void print() const`
Prints the heap in a rotated tree-like format:

```
      4
   7
      6
9
      4
   5
      2
```

Right subtree is displayed above the node, left subtree below it.

Example

```cpp
MinHeap<int> minHeap;
minHeap.insert(5);
minHeap.insert(3);
minHeap.insert(7);

int min = minHeap.peek(); // 3
int removed = minHeap.popRoot(); // 3

MaxHeap<int> maxHeap;
maxHeap.insert(5);
maxHeap.insert(3);
maxHeap.insert(7);

int max = maxHeap.popRoot(); // 7

minHeap.print();
```

Stack

Abstract data structure based on LIFO (Last In First Out) principle. Supports push and pop operations. In our case stack is implemented using linked list.

Trigger

`stack` | `stk`

Classes

One class `Stack` template gets `T` type as a template parameter. Uses `Node` structure to store elements.

Methods

`void push(T val)`
Pushes a new value to the top of the stack.

`T pop()`
Removes the top element from the stack and returns it.

`bool isEmpty()`
Returns `true` if the stack is empty, `false` otherwise.

`int size()`
Returns the number of elements in the stack.

`void print()`
Prints the stack in a reversed order:

```
Stack (size=3): 5 22 3
```

Example

```cpp
Stack<char> stack;

stack.push('a');
stack.push('c');
stack.push('b');

std::cout << stack.pop() << "\n"; // b
std::cout << stack.pop() << "\n"; // c

stack.isEmpty(); // false
stack.size(); // 1

stack.print();
```

Queue

Abstract data structure based on FIFO (First In First Out) principle. Supports enqueue and dequeue operations. In our case queue is implemented using linked list with head and tail pointers for efficient insertion and removal.

Trigger

queue | que

Classes

One class `Queue` template gets `T` type as a template parameter. Uses `Node` structure for linked list implementation.

Methods

`void enqueue(T data)`
Adds a new element to the back of the queue.

`T dequeue()`
Removes and returns the front element from the queue.

Throws:

- `std::out_of_range` if queue is empty

`bool isEmpty()`
Returns `true` if the queue is empty, `false` otherwise.

`int getSize()`
Returns the number of elements in the queue.

`void print()`
Prints the queue from front to back:

```
Queue (size=3): 1 2 3
```

Example

```cpp
Queue<int> q;

q.enqueue(1);
q.enqueue(2);
q.enqueue(3);

std::cout << q.dequeue() << "\n"; // 1
std::cout << q.dequeue() << "\n"; // 2

q.isEmpty(); // false
q.getSize(); // 1

q.print();
```

Deque

Abstract data structure that allows insertion and removal from both ends. Deque (double-ended queue) supports push and pop operations at both the front and back. In our case deque is implemented using doubly-linked list with head and tail pointers.

Trigger

`deque` | `dq`

Classes

One class `Queue` template gets `T` type as a template parameter. Uses `Node` structure with `prev` and `next` pointers for doubly-linked list implementation.

Methods

`void pushBack(T data)`
Adds a new element to the back of the deque.

`void pushForward(T data)`
Adds a new element to the front of the deque.

`T popBack()`
Removes and returns the element from the back of the deque.

Throws:

- `std::runtime_error` if deque is empty

`T popForward()`
Removes and returns the element from the front of the deque.

Throws:

- `std::runtime_error` if deque is empty

`void print()`
Prints the deque from front to back:

```
Deque (size=3): 1 2 3
```

Example

```cpp
Queue<int> dq;

dq.pushBack(2);
dq.pushForward(1);
dq.pushBack(3);

std::cout << dq.popForward() << "\n"; // 1
std::cout << dq.popBack() << "\n";    // 3

dq.print(); // Deque (size=1): 2
```