Built-in Data Structures (C++ STL Containers and Java Collections)

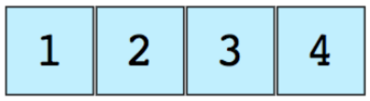
A data structure is a container that stores data in a specific layout. This "layout" allows a data structure to be efficient in some operations and inefficient in others. Your goal is to understand data structures so that you can pick the data structure that's most optimal for the problem at hand.

C++ STL (Standard Template Library) and **Java Collection and Java Map** template classes provide common programming data structures and related methods. Operations such as searching, sorting, insertion, manipulation, deletion, etc., can easily be performed by those data structures and methods. Those data structures are dynamic. They can grow and shrink at the run time.

Data can be structured in many ways but the commonly used practices are array, list, stack, queue, tree and hash table.

Array

An array is a linear data structure where each data element is assigned a positive numerical value called the *Index*, which corresponds to the position of that item in the array. C++ and Java define the starting index of the array as 0. Arrays can be one-dimensional (vector - as shown below) or multi-dimensional (matrix - arrays within arrays).



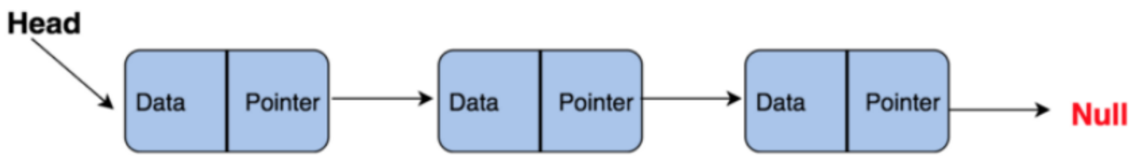
Basic Operations on Arrays

• Insert	: Inserts an element at given index	O(n)
• Get	**:** Returns the element at given index	O(1)
• Delete	: Deletes an element at given index	O(n)
• Size	: Get the total number of elements in array	O(1)
• Search	: Find the index of an element	O(n)
• Distance	: Number of elements between two positions	O(1)

Linked List

A linked list is another linear data structure which might look similar to arrays at first but differs in memory allocation, internal structure and how basic operations of insertion and deletion are carried out.

A linked list is like a chain of nodes, where each node contains information like data and a pointer to the succeeding node in the chain. In a double linked list, each node has two pointers, one for the next node and one for the previous node.



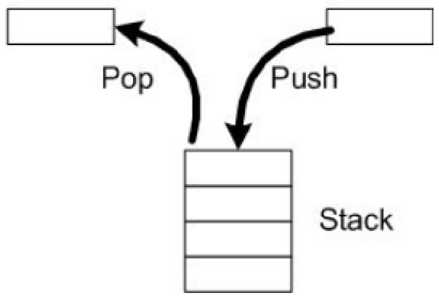
Basic operations of Linked List:

InsertAtEnd	: Inserts given element at the end of the linked list	O(1)
InsertAtHead	: Inserts given element at the start/head of the linked list	O(1)
Delete	: Deletes given element from the linked list	O(1)
DeleteAtHead	: Deletes first element of the linked list	O(1)
Search	: Returns the given element from a linked list	O(n)
isEmpty	: Returns true if the linked list is empty	O(1)
Get	: Returns the element at given index	O(n)
Distance	: Number of elements between two elements	O(n)

Stack

A stack is another linear data structure represented by a real physical stack or pile, a structure where insertion (push) and deletion (pop) of items takes place at one end called top of the stack. Stack is a *last in first out* structure (LIFO).

A real-life example of Stack could be a pile of books placed in a vertical order. In order to get the book that's somewhere in the middle, you will need to remove all the books placed on top of it. This is how the LIFO method works.



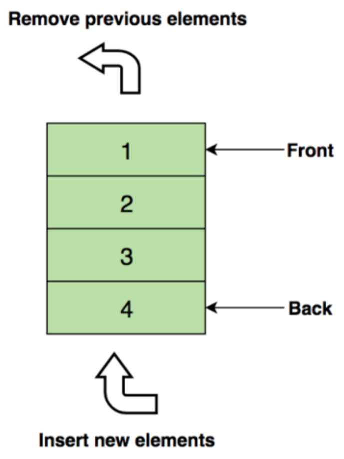
Basic operations of stack:

Push	: Inserts an element at the top	O(1)
Pop	: Returns the top element after removing from the stack	O(1)
isEmpty	: Returns true if the stack is empty	O(1)
Top	: Returns the top element without removing from the stack	O(1)

Queue

Similar to Stack, Queue is another linear data structure that stores the element in a sequential manner. The only significant difference between Stack and Queue is that instead of using the LIFO method, Queue implements the FIFO method, which is short for *First in First Out*.

A perfect real-life example of Queue: a line of people waiting at a ticket booth. If a new person comes, they will join the line from the end, not from the start — and the person standing at the front will be the first to get the ticket and hence leave the line.

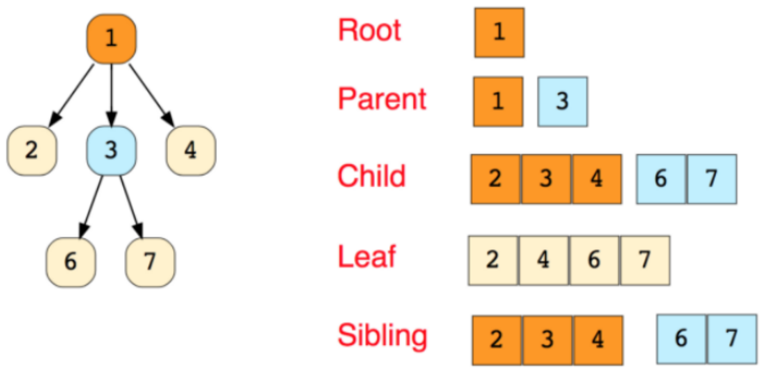


Basic operations of Queue

• Enqueue()	: Inserts element to the end of the queue	O(1)
• Dequeue()	: Removes an element from the start of the queue	O(1)
• isEmpty()	: Returns true if queue is empty	O(1)
• Top()	: Returns the first element of the queue	O(1)

Tree

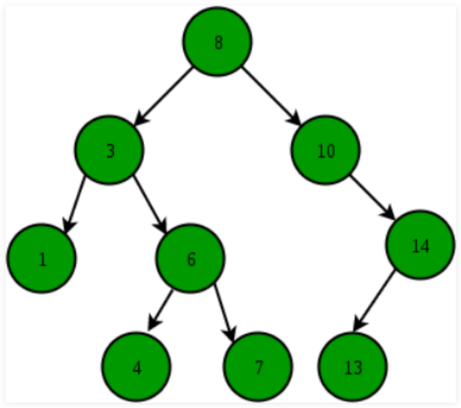
A tree is a hierarchical data structure consisting of vertices (nodes) and edges that connect them. Here's an image of a simple tree, and basic terminologies used in tree data structure:



Binary Search Tree

Binary Search Tree, is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Basic operations of Queue

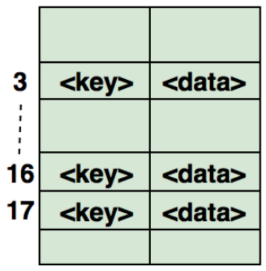
• Insert	: Inserts element	O(lgn)
• Erase	: Remove element	O(lgn)
• Find	: Find a target value	O(lgn)
• Max	: Returns max value	O(1)
• Min	: Return min value	O(1)
• Distance:	Number of elements between two elements	O(n)

Hash Table (Hash Map)

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.



C++ STL Containers

Sequence Containers

They implement data structures which can be accessed in a sequential manner.

- vector
- list (double linked list)
- deque (double ended queue with expansion and contraction on both the ends)
- arrays (a better alternative for C-style arrays)
- forward_list (single linked list, introduced in C++11)

Container Adaptors

They provide a different interface for sequential containers.

- queue
- priority_queue (the first element of the queue is the greatest/smallest of all elements in the queue)
- stack

Associative Containers

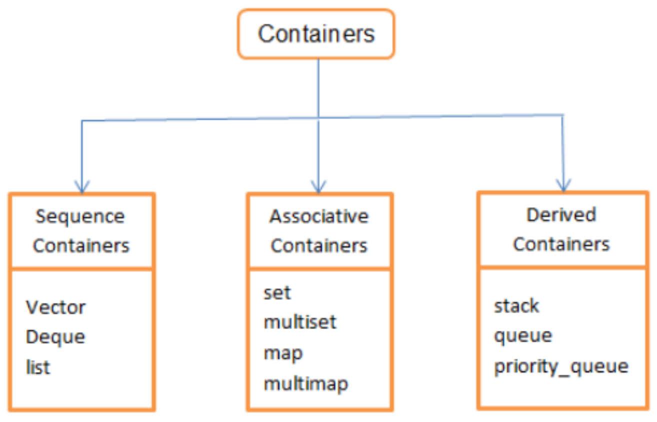
They implement sorted data structures (red-black tree) that can be quickly searched (O(log n) complexity).

- set (binary search tree)
- multiset (set with duplicate elements)
- map (each element has a key value and a mapped value, no duplicate key values)
- multimap (multiple elements can have the same key, can be implemented with a multiset)

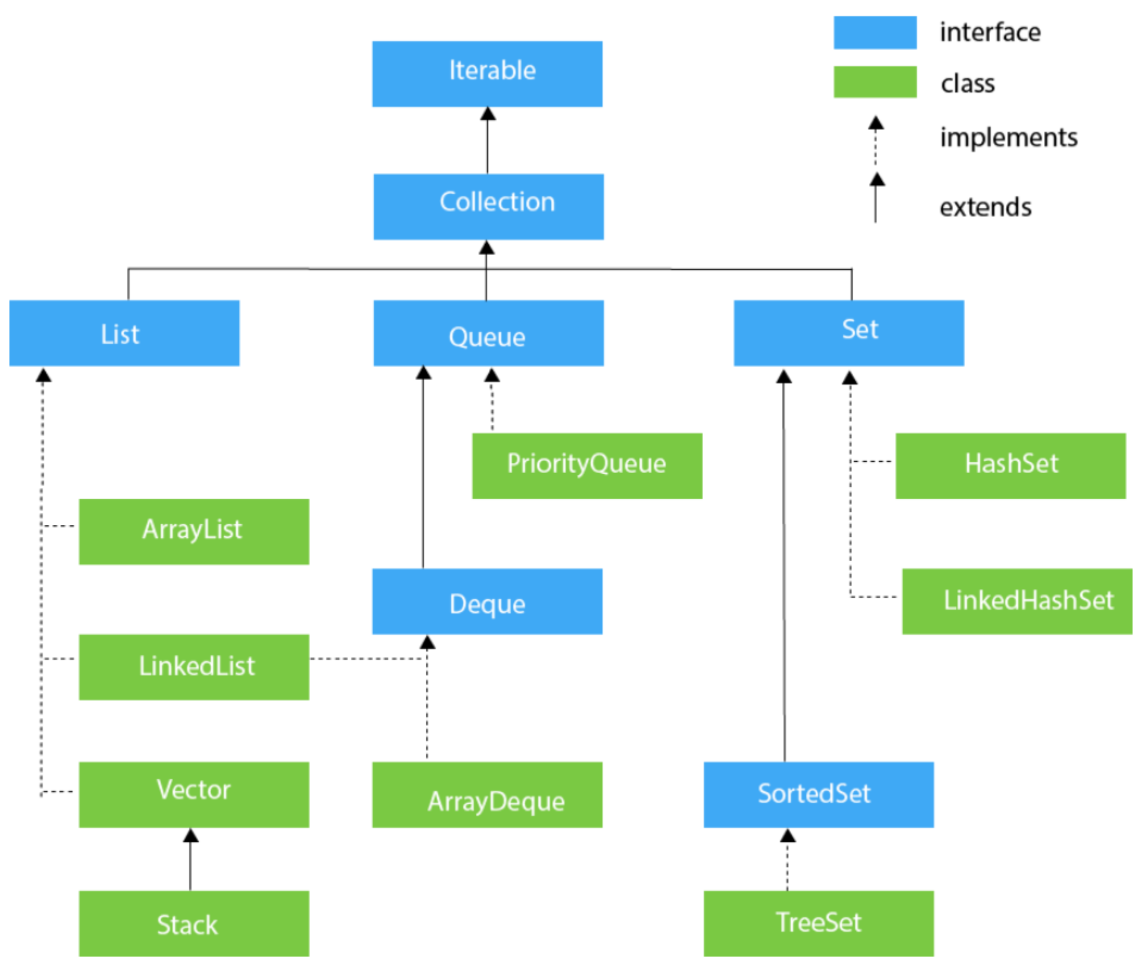
Unordered Associative Containers

They implement unordered data structures (hash table) that can be quickly searched (O(n) complexity). They occupy more space than their counterpart ordered structures.

- unordered_set (Introduced in C++11)
- unordered_multiset (Introduced in C++11)
- unordered_map (Introduced in C++11)
- unordered_multimap (Introduced in C++11)



Java Collections



Sample Program 1: The Balanced Parentheses Problem

The problem asks you to check if a string containing parentheses is "balanced", meaning is each open parenthesis "(", "[", or "{"met with/balanced out by a closed counterpart parenthesis ")", "]", or "}".

Sample Input 1

```
[[({()})]]
```

Sample Output 1

YES

Sample Input 2

```
[{({()})]}
```

Sample Output 2

NO

Solution:

To solve this problem we can parse the string from left to right, whenever we encounter with an close parenthesis we check if the last open parenthesis match with that parenthesis. If there is any mismatch the string is unbalanced. If there are some open parentheses left at the end again the string is balanced. To implement this idea we need a data structures to add the open parentheses in one end and remove the last element in the same end. This is a LIFO structure. We can employ a *stack* for this job.

Code:

```
// Balanced Parentheses.cpp : Stack implementation

#include <iostream>
#include <string>
#include <stack>

using namespace std;

string go(string str)
{
	stack<char> s;
	for (char ch : str)
	{
		//Open parenthesis. Push and continue.
		if (ch == '(' || ch == '[' || ch == '{')
		{
			s.push(ch);
			continue;
		}

		//There should be some open parentheses
		if (s.size() == 0)
			return "NO";

		char preCh = s.top();
		s.pop();

		//Check if there is a mismatch
		if ((ch == '(' && preCh != ')') || (ch == '[' && preCh != ']') || (ch == '{'
								&& preCh != '}'))
			return "NO";
	}

	//if stack is empty return "YES".
	return (s.size() == 0 ? "YES" : "NO");
}

int main()
{
	string str = "[[({()})]]";
	cout << go(str) << endl;
}
```

Output

```
YES
```

Exercise: Make alternative implementations of the “Balanced Parentheses” problem: 1) with vector, 2) with deque and 3) with recursion.

Sample Program 2: Add, Remove and Find

Three operations defined on a list: “Add”, “Remove” and “Find”. There are N consecutive operations. Output the result of each Find operation as an integer that is the occurrence of the target number.

The list is empty in advance. “Remove” operation removes only one instance of a value.

Sample Input	**Sample Output**
9	1
Add 3	2
Add 4	0
Add 3	1
Find 4	
Find 3	
Remove 5	
Remove 3	
Find 7	
Find 3	

To solve this problem, we need a data structure that supports duplicate values and performs add, remove an arbitrary element and find operations in an efficient way.

Structure	Add	Remove	Find	Duplicate
Vector	O(1)	O(n)	O(n)	Yes
List	O(1)	O(1)	O(n)	Yes
Stack	O(1)	No	No	Yes
Queue	O(1)	No	No	Yes
Set	O(lgn)	O(lgn)	O(lgn)	No
Multi Set	O(lgn)	O(lgn)	O(lgn + m)	Yes
Map	O(lgn)	O(lgn)	O(lgn)	Yes
Unordered Map	O(1)	O(1)	O(1)*	Yes

Multiset and Map performs all three operations in O(lgn) time, unordered map performs in O(1) time. Unordered_map consumes extra memory for internal hashing and Find time is input sensitive. It may increase to O(n).

Code 1: Multiset Implementation

```
// Add Remove Find Multiset.cpp :

#include <iostream>
#include <set>
#include <string>
#include <vector>

using namespace std;

int n=9;
vector<pair<string, int>> input = { {"Add",3},{"Add",4}, {"Add",3}, {"Find",4}, {"Find",3},
                                    {"Remove", 5}, {"Remove", 3},{"Find",7},{"Find",3} };

int main()
{
        multiset<int> ms;

        for (int i = 0; i < n; i++)
        {
                string op = input[i].first;
```

```
			int num = input[i].second;
			if (op == "Add")
				ms.insert(num);
			else if (op == "Remove")
			{
				//use iterator to point a single element
				auto it = ms.find(num);
				if (it != ms.end())
					ms.erase(it);
			}
			else //if (op == "Find")
			{
				count(num) << endl;cout << ms.
			}
		}
}
```

Output

```
1 2 0 1
```

Code 2: Map Implementation

```
// Add Remove Find Map.cpp :

#include <iostream>
#include <map>
#include <string>
#include <vector>

using namespace std;

int n=9;
vector<pair<string, int>> input = { {"Add",3},{"Add",4}, {"Add",3}, {"Find",4}, {"Find",3},
                                      {"Remove", 5}, {"Remove", 3},{"Find",7},{"Find",3} };

int main()
{
	map<int, int> m;

	for (int i = 0; i < n; i++)
	{
		string op = input[i].first;
		int num = input[i].second;
		if (op == "Add")
			m[num]++;
		else if (op == "Remove")
		{
			if (m[num] > 0)
				m[num]--;
		}
		else //if (op == "Find")
			cout << m[num] << endl;
	}
}
```

Output

```
1 2 0 1
```

Exercise: Use a frequency vector to speed up algorithm of the “Add, Remove and Find” problem.

Sample Program 3: Maze

Maze problems usually have DFS and BFS solutions. If you need to print only the shortest path from the entrance to the exit you should use BFS, if you need to print all the paths form the entrance to the exit, you should use DFS. DFS can be implemented with recursion or with Stack. The best way to implement BFS is to use a Queue. If you need to print any path (not necessary to be the shortest one) you can use “Right Hand Rule” for a regular maze. Imagine that you are walking through the maze all the time touching the wall on your right with your right hand. Keep going until you reach the exit.

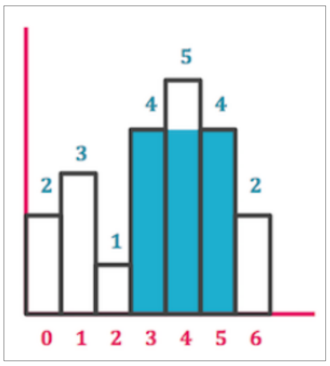
Code: BFS solution

```
// Maze Queue.cpp : Shortest path wiht BFS

#include <iostream>
#include <vector>
#include <queue>

using namespace std;

//0 empty, 1 wall. enter is left upper corner,
//exit is bottom right corner.
int N = 5;
vector<vector<int>> maze = { {0,1,0,1,0},
                             {0,0,0,0,0},
                             {0,1,0,1,0},
                             {0,1,0,1,0},
                             {0,0,0,0,0}
                         };

//up, left, down, right
vector<int> xDir = { -1, 0, 1, 0 };
vector<int> yDir = { 0, 1, 0, -1 };

//return true if p is inside the maze.
bool inBorder(pair<int, int> p)
{
       if (p.first < 0 || p.first >= N)
              return false;
       if (p.second < 0 || p.second >= N)
              return false;
       return true;
}

//return true if p is inside the maze
//and that position is 0.
bool valid(pair<int, int> p)
{
       return inBorder(p) && maze[p.first][p.second] == 0;
}

//Traverse the path from exit to start and
//print it in reverse order.
//This function can be implemented with a stack.
void printPath(pair<int,int> p)
{
       int x = p.first;
       int y = p.second;
       if (x != 0 || y != 0)
       {
              for (int i = 0; i < 4; i++)
              {
                     pair<int, int> newPos = { x + xDir[i], y + yDir[i] };
                     if (!inBorder(newPos))
                            continue;

                     if (maze[x + xDir[i]][y + yDir[i]] == maze[x][y] - 1)
                     {
                        printPath(newPos);

                        break;
                    }
                }
            }
        cout << "(" << x << "," << y << ") ";
}

int main()
{
        queue<pair<int, int>> q;
        q.push(make_pair(0,0));
        maze[0][0] = 2; //0 and 1 are already in the maze
        while (!q.empty())
        {
                pair<int, int> pos = q.front();
                q.pop();

                //We have the solution
                if (pos == make_pair(N - 1, N - 1))
                {
                        printPath(pos);
                        break;
                }

                //try to expand to each neighbour
                for (int i = 0; i < 4; i++)
                {
                        pair<int, int> newPos = pos;
                        newPos.first += xDir[i];
                        newPos.second += yDir[i];
                        if (valid(newPos))
                        {
                                maze[newPos.first][newPos.second] = maze[pos.first][pos.second] + 1;
                                q.push(newPos);
                        }
                }
        }
}
```

Output

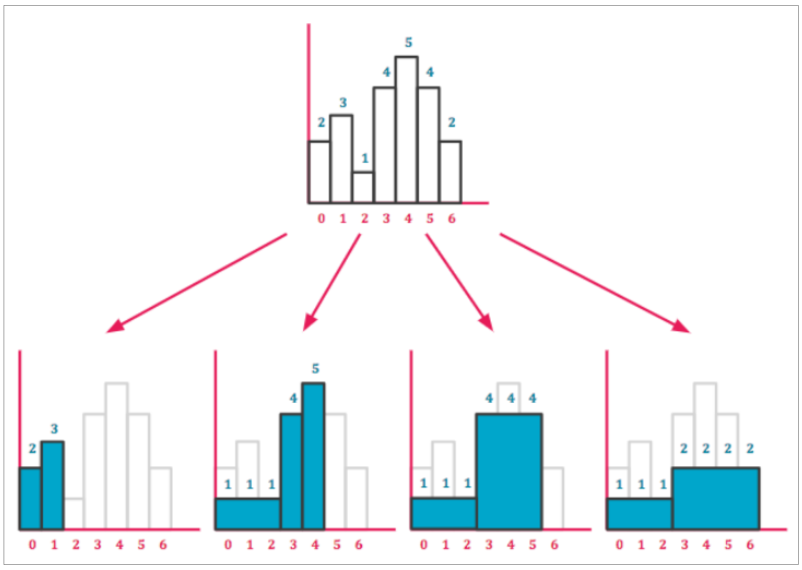
```
(0,0) (1,0) (1,1) (1,2) (1,3) (1,4) (2,4) (3,4) (4,4)
```

Sample Program 4: Largest Rectangular Area in a Histogram



This problem can be solve in O(n*n) time with brute force, in O(nlgn) time with divide and conquer and in O(n) time with a Stack.

To find the sequence of bars with climbing height, we traverse all bars from left to right and hold the index of bars with bigger height in a Stack. A bar is popped from the Stack when a bar of smaller height is seen; meanwhile, we calculate the largest rectangular area in the sequence because the sequence with climbing height ends here.



Code: Stack Solution

```
// Rectangle Histogram.cpp : Stack Solution

#include<iostream>
#include <vector>
#include<stack>
#include <algorithm>

using namespace std;

int N = 7;
vector<int> bars = { 2,3,1,4,5,4,2 };
//--------------------------------------------
int go()
{
	//s stores indices of bars in non-decreasing order
	stack<int> s;

	int tempRes, res = 0;

	// Proceeds bars one by one
	int i = 0;
	while (i < N)
	{
		// current bar is not smaller than the last added bar in stask
		if (s.empty() || bars[s.top()] <= bars[i])
		{
			s.push(i++);
			continue;
		}

		//current bar is smaller. Pop the bars in the stack
		//and calculate are for the poped bar until the current
		//bar is not shorter.

		int sTop = s.top();  //index of top element
		s.pop();
```

```
            if (s.empty())
                tempRes = bars[sTop] * i ;
            else
                tempRes = bars[sTop] * (i - s.top() - 1);

            res = max(res, tempRes);
        }

        // Now pop the remaining bars from the stack and calculate
        // area for each popped bar.
        while (!s.empty())
        {
            int sTop = s.top();
            s.pop();
            if (!s.empty())
                tempRes = bars[sTop] * (i - s.top() - 1);
            else
                tempRes = bars[sTop] * i;

            res = max(res, tempRes);
        }
        return res;
}
//-------------------------------------------
int main()
{
        cout << go() << endl;
        return 0;
}
```

Output

```
12
```

Hint: Adding a virtual first bar with zero height into the histogram at the beginning of the program will prevent the stack from being empty.

Exercise: Solve the “Largest Rectangular Area in a Histogram” problem with divide and conquer method in O(nlgn) time.

Sample Program 5: Next Greater Element

Print the next greater element of all elements in an array of integer. If there is no greater element then print 99.

Sample Input

```
9
1 2 -3 1 7 -2 -3 3 6
```

Sample Output:

```
2 7 1 7 99 3 3 6 99
```

Algorithm

This problem can be solved in O(n) time and O(n) space by using a stack.

We traverse the array once.

1. If the stack is empty or the current element is smaller than top element of the stack, then push the current element (index of it) on the stack.
2. If the current element is greater than top element of the stack, then this is the next greater element of the top element. Keep popping elements from the stack till a larger element than the current element is found on the stack or till the stack becomes empty. Push the current element on the stack.
3. Repeat steps 1 and 2 till the end of array is reached.
4. Finally pop remaining elements from the stack and print null for them.

Please note that at any instance, the stack will always be in sorted order having least element at the top and largest element at the bottom.

Code:

```
// Next Greater Element Stack.cpp : O(N)

#include <iostream>
#include <vector>
#include <stack>

using namespace std;

int main()
{
	vector<int> numbers = { 1,  2, -3,  1,  7, -2, -3,  3,  6 };
	int N = numbers.size();
	vector<int> nextGreater(N);
	stack<int> s;

	//Find the next greater element of each element.
	for (int i=0; i<N; i++)
	{
		int curNum = numbers[i];
		while (!s.empty() && curNum > numbers[s.top()])
		{
			nextGreater[s.top()] = curNum;
			s.pop();
		}
		s.push(i);
	}

	//Remaining elements in the stack.
	//They do not have next greater element.
	while (!s.empty())
	{
		nextGreater[s.top()] = 99;
		s.pop();
	}

	for (int x : nextGreater)
		cout << x << " ";
}
```

Output:

```
2 7 1 7 99 3 3 6 99
```

Sample Program 5: Maximum of all Subarrays of Size K.

Given an array and an integer K, find the maximum for each and every contiguous subarray of size k.

Sample Input

```
9 3
1  2  3  1  4  5  2  3  6
```

Sample Output:

```
3 3 4 5 5 5 6
```

Solution 1 (Sliding Window): Set a sliding window with the length K, move it from the beginning of the array to the end, and calculate the maximum element for each window position. This solution runs in O(nk) time.

Solution 2 (Set): We can keep the current numbers of the sliding window in a multiset. Removing the left most element and adding a new element will take O(k) time and finding the max element O(1) time. Overall time complexity will be O(nlgk).

Solution 3 (Decomposition): Decompose the array into block of size k. For each element find the maximum value in its block from left-most element until that element (leftMax), and from right-most end until that element (rightMax). The maximum value of each window (not block) is the maximum of its leftMax of right-most element and rightMax of its left-most element. Every step can be performed in a linear time. Total time complexity is O(n).

Solution 4 (Stack): For each position find the index of the next greater element in O(n) time using a stack. The maximum value in a window is the first element its next greater index is not in that window. Time complexity is O(n)

Solution 5 (Deque): Instead of keeping K elements for a window, we can just keep the useful elements. If the new number greater than some elements in a window, those smaller elements become useless. Maintain the invariant of the *deque* so that front element is the maximum, and back element is the minimum. *Add* and *Remove* processes occur at the back of the *deque*. Store indices of the numbers in the *deque* instead of actual values to handle the window size.

Code: Deque Solution

```
// Maximum of each subarray with K elements.
// Deque implementation. O(n)

#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int N,K;
vector<int> numbers;
//-------------------------------------
void go()
{
	deque<int> dq;
	//Process the first K numbers
	dq.push_front(0);
	int i = 1;
	for (; i < K; i++)
	{
		//Remove the numbers in the deque if they are smaller
		//than the current number
		while (!dq.empty() && numbers[dq.back()] <= numbers[i])
			dq.pop_back();
			dq.push_back(i);
		}

		//Process the remaining numbers
		for (; i < N; i++)
		{
			//Front of the deque is maximum of current window
			cout << numbers[dq.front()]<<" ";

			//window size is K+1 remove front elemen (min index)
			if (i >= dq.front() + K)
				dq.pop_front();

			//Remove the numbers in the deque if they are smaller
			//than the current number
			while (!dq.empty() && numbers[dq.back()] <= numbers[i])
				dq.pop_back();
			dq.push_back(i);
		}
		//print maximum of the last window
		cout << numbers[dq.front()];
}
//-------------------------------------
int main()
{
		numbers = { 7, 5, 6, 2, 4, 3, 5, 1, 6, 7};
		N = numbers.size();
		K = 3;
		go();
}
```

Output:

```
7 6 6 5 6 7
```

Sample Program 7: Cow Dance (PQ)

A stage of size K can support K cows dancing simultaneously. The N cows in the herd (1≤N≤10,000) are conveniently numbered 1…N in the order in which they must appear in the dance. Each cow i plans to dance for a specific duration of time d(i). At time 0, cows 1…K appear on stage and start dancing. When the first of these cows completes her part, she leaves the stage and cow K+1 immediately starts dancing, and so on, so there are always K cows dancing (until the end of the show, when we start to run out of cows). The show ends when the last cow completes her dancing part, at time T. Make a program to calculate minimum T.

(Modified from the Usaco Jan 2017 Cow Dance Show problem.)

Sample Input

8 4

4 7 2 6 9 8 1 8 4 2

Sample Output

15

To solve this problem, we need a data structure that performs two operations efficiently: 1) remove the earliest ending time of K cows on the stage (min value), 2) add a new cow. Priority Queue and multi set perform both operations in lg(n) time.

Code: Priority Queue

```
// Cow Dance.cpp : Priority Queue. O(nlgk) time

#include <iostream>
#include <vector>
#include <queue>

using namespace std;

int main()
{
	vector <int> cows = { 4, 7, 2, 6, 9, 8, 1, 8, 4, 2 };
	int N = cows.size();
	int K = 4;

	//min heap stores ending time of cows
	priority_queue<int, vector<int>, greater<int>> pq;

	//push the first K cows into the pq
	for (int i = 0; i < K; i++)
		pq.push(cows[i]);

	//move a cow from the stage and add the new one
	for (int i = K; i < N; i++)
	{
		int cow = pq.top();
		pq.pop();
		cow += cows[i];
		pq.push(cow);
	}

	//There are last K cows on the stage
	//Keep poppint them until the last cow.
	while (pq.size() > 1)
		pq.pop();

	//Print the ending time of the last cow
	cout << pq.top() << endl;
}
```

Output:

```
15
```

Practice Problems

1. **Arithmetic Expression:** Make a program that evaluates an arithmetic expression that contains integers, parentheses, and multiplication, addition and subtraction operators. 5*(10 –(4+5*2)) = ?
2. **Maze with Stack:** Make a program that prints all possible paths from the entrance to the exit in a maze. This problem can be solved with recursion and with stack. Implement your solution with stack.
3. **Counting Inversions:** Inversions in an array can be counted with divide and conquer method in O(nlgn) time. How can you solve this problem in O(n*n) using a multiset?
4. **Next Greater Element:** Find the next greater element of each element in an circular array.
5. **Largest Rectangular Area:** Make a program to find the largest rectangular area of 1s in a binary matrix. Use the largest rectangular area in a histogram idea to solve this problem.
6. **Subarrays with sum K:** Calculate number of subarrays their sum exactly equal to K.

7. **Stack with Max Value:** Implement a custom stack structure that retrieves the max value in the stack in O(1) time. (Hint: Use two stacks)
8. **Implement Stack using PQ:** How can you implement stack using a priority queue.

Useful Links and References

https://opensourceforu.com/2015/06/data-structures-made-easy-with-java-collections/
https://www.freecodecamp.org/news/the-top-data-structures-you-should-know-for-your-next-coding-interview-36af0831f5e3/
https://en.wikibooks.org/wiki/Data_Structures
https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/
https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/
https://www.geeksforgeeks.org/collections-in-java-2/
https://www.geeksforgeeks.org/the-c-standard-template-library-stl/
https://www.javatpoint.com/collections-in-java
https://tech.pic-collage.com/algorithm-largest-area-in-histogram-84cc70500f0c
https://discuss.codechef.com/t/sliding-window-maximum-maximum-of-all-subarrays-of-size-k-help/27198
https://www.ideserve.co.in/learn/next-great-element-in-an-array