Header files

- Forward declarations
- Eg: iostream
- Syntax: #include <iostream>

- Cout<
- Cin >

Namespace

- The namespace provides a scope region (called namespace scope) to the names declared inside of it
- C++ has all of the functionality in the standard library into a namespace named “std” (short for standard)
- Eg: std::cout, std::cin
- Syntax: `using namespace std` (access without std prefix)

a.cpp:

```cpp
#include <iostream>

void myFcn(int x)
{
    std::cout << x;
}
```

main.cpp:

```cpp
#include <iostream>

void myFcn(int x)
{
    std::cout << 2 * x;
}

int main()
{
    return 0;
}
```

```
#include <iostream>
using namespace std;

int main() {
    double n1, n2, n3;
    cout << "Enter three numbers: ";
    cin >> n1 >> n2 >> n3;
    // check if n1 is the largest number
    if(n1 >= n2 && n1 >= n3)
        cout << "Largest number: " << n1;
    // check if n2 is the largest number
    else if(n2 >= n1 && n2 >= n3)
        cout << "Largest number: " << n2;
    // if neither n1 nor n2 are the largest, n3 is the largest
    else
        cout << "Largest number: " << n3;

    return 0;
}
```

Structures

Structures work just like in c

```
struct Fraction
{
	int numerator;
	int denominator;
};

// Now we can make use of our Fraction type
int main()
{
	Fraction f;
// this actually instantiates a Fraction
// object named f
	f.numerator = 3;
	f.denominator = 4;

	return 0;
}
```

Object Oriented Programming - Prelude

```
#include <iostream>
struct Date
{
    int day{};
    int month{};
    int year{};
};
void printDate(const Date& date)
{
    std::cout << date.day << '/' << date.month << '/' << date.year; // assume DMY format
}
int main()
{
    Date date{ 4, 10, 21 }; // initialize using aggregate initialization
    printDate(date);        // can pass entire struct to function
    return 0;
}
```

Object Oriented Programming

```
#include <iostream>
class Date       // we changed struct to class
{
public:          // and added this line, which is called an access specifier
    int m_day{}; // and added "m_" prefixes to each of the member names
    int m_month{};
    int m_year{};
};
void printDate(const Date& date)
{
    std::cout << date.m_day << '/' << date.m_month << '/' << date.m_year;
}
int main()
{
    Date date{ 4, 10, 21 }; // date is an object of Date class
    printDate(date);
    return 0;
}
```

Access modifiers

Specifiers	Same Class	Derived Class	Outside Class
`public`	Yes	Yes	Yes
`private`	Yes	No	No
`protected`	Yes	Yes	No

Object Oriented Programming

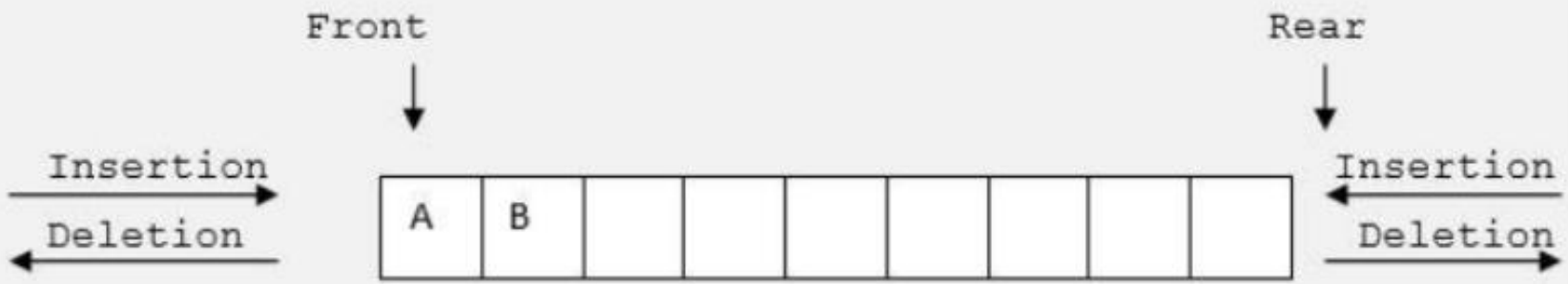
```
class Date {
// Any members defined here would default to private
public: // here's our public access specifier
    void print() const // public due to above public: specifier
    {
        // members can access other private members
        std::cout << m_year << '/' << m_month << '/' << m_day;
    }

private: // here's our private access specifier
    int m_year { 2020 };  // private due to above private: specifier
    int m_month { 14 }; // private due to above private: specifier
    int m_day { 10 };   // private due to above private: specifier
};
int main() {
    Date d{};
    d.print();
// okay, main() allowed to access public members
    return 0;
}
```

Double Ended Queue (dequeuer)

Description:

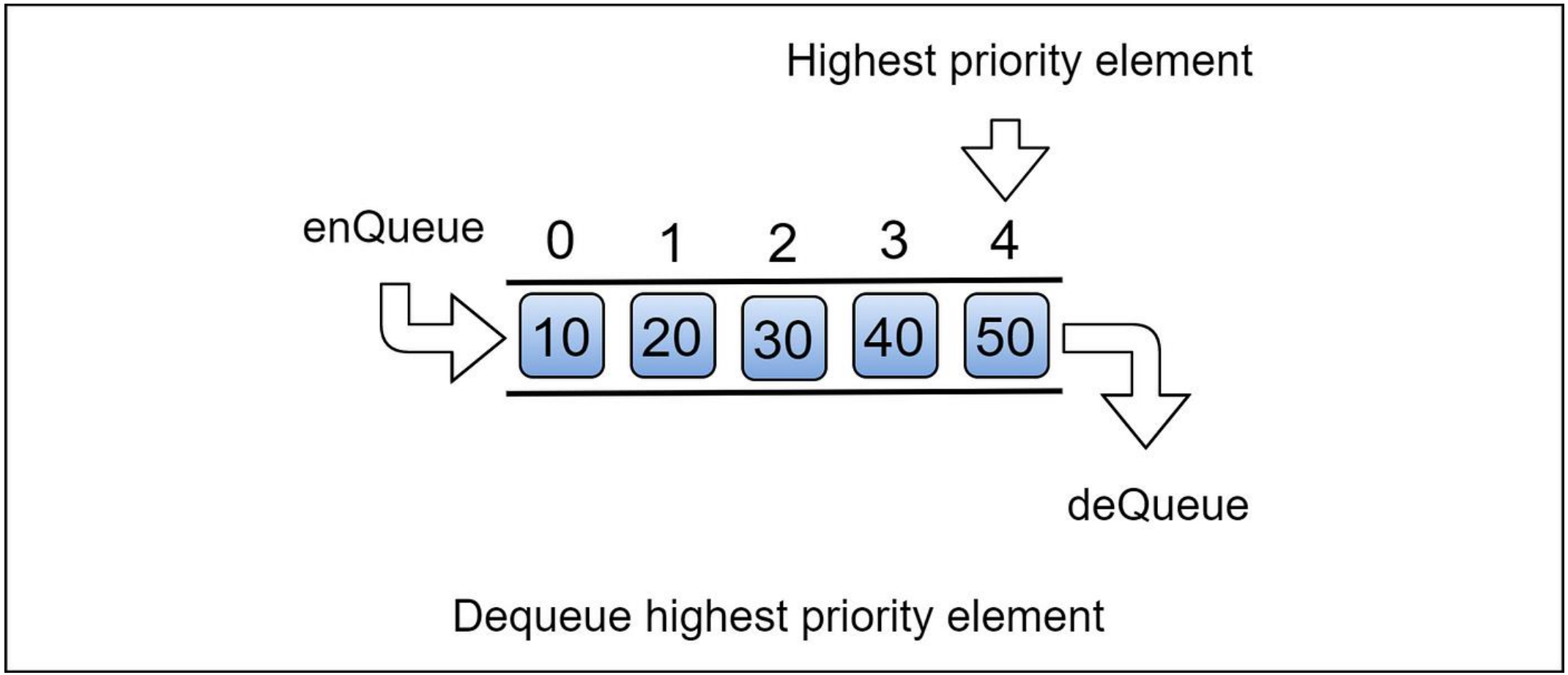
A queue that supports insertion and deletion at both the front and rear is called **double-ended queue or Dequeue.** A Dequeue is a linear list in which elements can be added or removed at either end but not in the middle.



The operations that can be performed on Dequeue are

1. Insert to the beginning
2. Insert at the end
3. Delete from the beginning
4. Delete from end

Priority queue



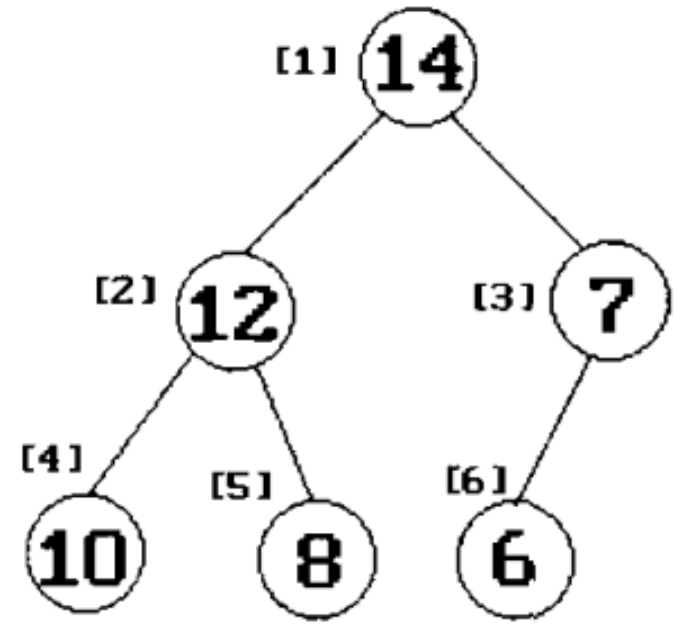
structure *MaxHeap* is

objects: a complete binary tree of $n > 0$ elements organized so that the value in each node is at least as large as those in its children

functions:

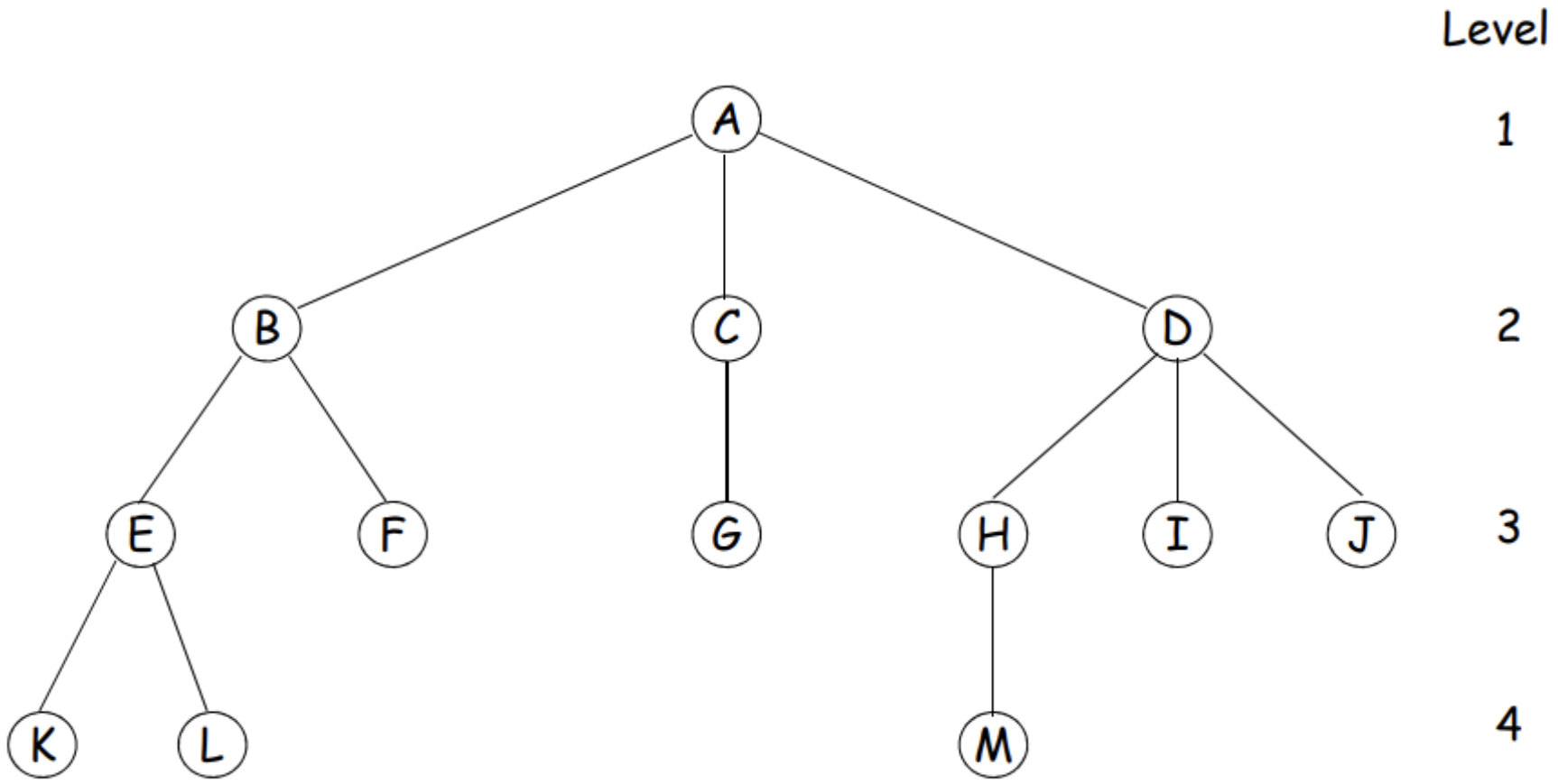
for all *heap* ∈ *MaxHeap*, *item* ∈ *Element*, *n*, *max_size* ∈ integer

MaxHeap Create(*max_size*)	::=	create an empty heap that can hold a maximum of *max_size* elements.
Boolean HeapFull(*heap*, *n*)	::	**if** (*n* == *max_size*) **return** *TRUE* **else return** *FALSE*
MaxHeap Insert(*heap*, *item*, *n*)	::=	**if** (!HeapFull(*heap*, *n*)) insert *item* into *heap* and return the resulting heap **else return** error.
Boolean HeapEmpty(*heap*, *n*)	::	**if** ($n > 0$) **return** *TRUE* **else return** *FALSE*
Element Delete(*heap*, *n*)	::=	**if** (!HeapEmpty(*heap*, *n*)) **return** one instance of the largest element in the heap and remove it from the heap **else return** error.



Structure 5.2: Abstract data type *MaxHeap*

TREES



STL - Maps

- Data stored as key value pairs
- Implemented using red-black tree (self balancing tree)
- Can be – unordered and ordered based on keys
- Time complexity for **ordered** maps of:
- Searching – O(logn)
- Insertion – O(logn)
- Deletion – O(logn)

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main() {
    map<int, string> sample_map;
    sample_map.insert(pair<int, string>(1, "one"));
    sample_map.insert(pair<int, string>(2, "two"));
    sample_map.insert(pair<int, string>(4, "four");
    sample.map[5] = "five";

    cout << sample_map[1] << " " << sample_map[2] << endl;

    return 0;
}
```

```
int main() {
    map<int, string> sample_map { { 1, "one"}, { 2, "two" } };
    sample_map[3] = "three";
    sample_map.insert({ 4, "four" });

    // accessing method 1
    map<int, string>::iterator it;
    for (it = sample_map.begin(); it != sample_map.end(); it++) {
        cout << it->second << " ";
    }
        cout << endl;

    // accessing method 2
    for (auto& entry : sample_map) {
        cout << entry.second << " ";
    }
    cout << endl;
    return 0;
}
```

STL – Maps methods

- map.find()
  - The find operation is used to search for a particular key in the map.
  - Time Complexity: O(log n)
- map.erase()
  - The erase operation is used to remove elements from the map based on their keys or iterators.
  - Time Complexity: O(log n)

STL – Unordered Maps

- Data stored as key value pairs
- Implemented using hash tables
- Can be – unordered and ordered based on keys
- Time complexity of:

  Searching – O(1) ,i.e, map.find()

  Insertion – O(1) ,i.e, map.insert()

  Deletion – O(1) ,i.e, map.erase()
- Declared as *unordered_map<>*

STL - Set

- Stores unique elements like in maps
- Implemented using red-black tree (self balancing tree)
- Can be – unordered, ordered, or multiset
- Time complexity of:
  Searching – O(logn)
  Insertion – O(logn)
  Deletion – O(logn)
- Can’t use set like set_name[i]

STL – Sets methods

- set_name.find()
  - The find operation is used to search for a particular key in the map.
  - Time Complexity: O(log n)
- set_name.erase()
  - The erase operation is used to remove elements from the map based on their keys or iterators.
  - Time Complexity: O(log n)

STL - Multiset

- Duplicates allowed
- Used to implement priority queues

```
multiset<string> s;
s.insert(“abc”);
s.insert(“abc”);
```