Big Five

IntCell is a class with a private parameter int storedValue

Destructor
~IntCell( )
{ delete storedValue; }

Copy constructor
IntCell(const IntCell & rhs)
{ storedValue = new int{ *rhs.storedValue }; }

Move constructor
IntCell(IntCell && rhs) : storedValue{ rhs.storedValue }
{ rhs.storedValue = nullptr; }

Copy assignment
IntCell & operator= (const IntCell & rhs) {
if(this != &rhs)
*storedValue = *rhs.storedValue; }

Move assignment
IntCell & operator= (IntCell && rhs) {
std::swap(storedValue, rhs.storedValue);
return *this; }

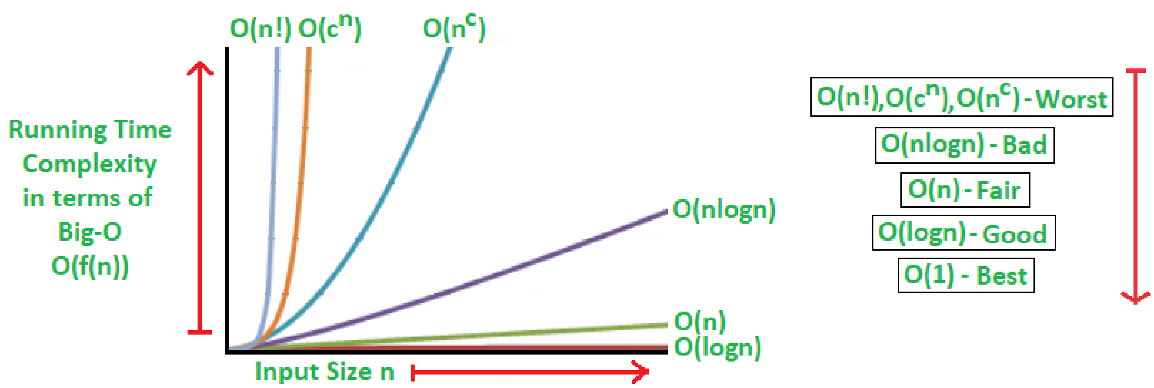
Examples of Big Five Usage

1. IntCell A{10}; **Construct qwee wwqe qwwor with one parameter**
2. IntCell B{A}; **Copy constructor**
3. IntCell B=A **Copy constructor**
4. IntCell X = A; **Copy constructor**
5. IntCell C; **Constructor with no parameters**
6. C=A; **Copy assignment**
7. D = new IntCell; **Constructor**
8. delete D; **Destructor**
9. IntCell A{move(B)}; **Move constructor**
10. X = move(A); **Move assignment**

Big O Notation: the worst-case scenario of an algorithm

Typical growth rates

c	constant time
log(N)	logarithmic
$\log^2(N)$	log-squared
N	linear
Nlog(N)	
N^2	Quadratic
N^3	Cubic
2^n	Exponential



STL Vector
- Constant time indexing
- Fast to add data at the end / slow at the front
- Slow to add data in the middle

STL List
- Implemented as a double linked list
- Fast insertion at any position
- No indexing

Iterators (STL)
- iterator begin(); // first item
- iterator end(); // position after last item

const_iterators
- cannot change values of iterator, but can read

Parameter passing
- **Call by value** [double a, double b]
    - Objects will not be changed by the function
- **Call by reference** [double &a, double &b]
    - Objects can be changed by the function
- **Call by constant-reference** [const double &a, const double &b]
    - Large objects cannot be changed by the function and will be the most expensive to copy

L and R Values
- L Values - objects that occupy location in memory. Are not temporary.
- R Values - temporary values OR values not associated with any object(a literal constant)

• Factorial recursive running time:

T(n) = 1 + T(n-1) for n >1, T(1)=2

T(n) = 1 + T(n-1)

= 1 + (1 + T(n-2)) = … =

= 1+(1(+…(1+T(n-k))..) (k 1's)

=> T(n) = k + T(n-k)

= (n-1) + T(n-(n-1))

= (n-1) + T(1)

= (n-1)+2

= n+1

=> T(n) = n + 1 => T(n) = O(n).

```
long Factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * Factorial(n – 1); }
```

Binary Search

If N is a power of 2(i.e. N = 2^k with k = log(N))

Then T(N/2^k) = log(N)

Fibonacci (bad example):

• Running time: T(n) = T(n-1) + T(n-2) + 2, T(0)=T(1)=1

• Since fib(n) = fib(n-1)+fib(n-2), we can prove (by induction) that T(n) >= fib(n)

• Section 1.2.5 proves that T(n) < $(5/3)^n$

=> T(n) =? (($(5/3)^n$)

Exponential: Really bad result! (Use iteration instead)

```
long Fibonacci(int n) {
    if (n <= 1)
        return 1;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2); }
```

Maximum SubSequence

Three ways to solve: triple loop, double loop or recursive

Triple: O(N^3) | *Double: O(N^2)* | *Recursive: O(Nlog(N))*

Recursive splits up sequence into half, finds max of that

Then compares left & right sum with half of the sequence

Tree Terms

- **Root:** start of tree
- **Child:** root of each subtree(r)
- **Parent:** r of each subtree root
- **Leaves:** node with no children

Depth: Length of path from root to node

Height: length of Longest path from node to leaf, H(tree) = H(root)

Internal Path Length Avg Case Analysis:

D(N) = D(i) + D(N - i -1) + (N - 1)

Types of Trees

- **Expression Trees:** equation based tree
- **Binary Trees:** each parent has ONLY 2 children
- **Binary Search Trees:** left/right comparison
- **AVL Trees:** balanced binary tree
- **Splay Trees:** self adjusts for balance when searching
- **B-Trees:** balanced trees (can only be a diff of +- 1 height)
- **M-ary Tree:** M children per node
- **Sets/Maps:** containers using keys in sorted order
- **Threaded Binary Tree:** leafs are pointed to predecessors of nodes | single: right leaf | double: both leaves

Facts for Trees

Binary trees worse case is inserting a **sorted list** of items

AVLTrees ensures that depth is **O(log(N))**

Amortized Cost of an AVLTree for insert/delete/find is **O(M*N),** where each operation is O(N) time

Splay trees can be deep but are balanced for every access using **zig-zag** / **zig-zig** rotation | will turn it into O(Mlog(N))

zig rotation: single **right rotation** in AVL **|** **zag rotation**: single **left rotation** in AVL

B-trees are good for large data storage, for storing into disks

Hashing

- Array of fixed size
- Implementation of hash
- Used for insert, delete and find in O(1) avg time
- **Random collision resolution:** 1/(1-λ) | empty probability: 1 - λ = p | non-empty: λ = 1 - p
- **Open addressing**
  - Linear probing: if the item is in the key slot, move onto the next slot till there is an empty slot
    - Hits: ½(1+1/1-λ) | Misses & Insert: ½(1+1/$(1-λ)^2$)
  - Quadratic probing: uses quadratic for rehashing
  - Double hashing: uses two hashes, hash once, then hash again
- **Closed addressing**
  - Seperate chaining: uses linked lists for collisions