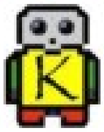
CS31 Study Guide: pointers, cstring, Arrays, string, cctype, functions and parameters, if statements, for/while/dowhile loops, iostream, variables.

DON'T FORGET THE SEMICOLONS



```
switch(intvars)
{
    case exp_1;
    case exp_2;
        break;
    case exp_3;
        break;
    default;
}
```

```
if(cond){…}
else if(cond) {…}
else{…}
```

ASCII TABLE INFO:

0=null	65-90=A-Z
32=space	97-122=a-z
48-57=0-9	10=new line

```
#include <cstring>
using namespace std;
char s[100] = "" //explicit empty init
char t[9] = "Hello"
//end of cstring is '\0' (null char)
strcpy(destination, source);

cout << t;    //output
cin.getline(s, 100); //input
cin.get(s)

int l = strlen(t); //string size
strcat(s "!!!"); //concatenation
// t<s compares memory locations
int c = strcmp(t,s); //comparison
/*
c<0 if t<s; c=0 if t==s; c>0 if t>s
*/
```

```
/* continue jumps to next
iteration of loop. break
leaves the loop entirely*/
```

```
for(int i=0; i<n; i++)
{ }
while(condition) { }
do{ }
while(condition);
```

```
#include <iostream>
cout << "hi" << endl;
int i;
cin >> i;
cin.ignore(10000, '\n');

//fixed decimal numbers
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
```

OMNOMNOMNOMNOMNOM

Variable Types

int
double
char
(float)

(long)
string
bool

```
/*parameters are
USUALLY passed by
value.
<Type>& is pass by
reference.
Arrays are always
by reference */
```

PACE YOURSELF

```
#include <string>
using namespace std;
string t; //empty string
string s = "hello";
string t = s; //assignment

cout << t[k];  //prints with cout
getline(cin, t); //instead of cin

int l = t.size() //string size
uses +-; //to concatenate/remove
bool b = t<=s; // comparison
```

```
Character Manipulation:
char nextSymbol;
cin.get(nextSymbol); //input char
cout.put(nextSymbol);//output char
```

```
/* variables only exist in the scope
    they are declared in! */
```

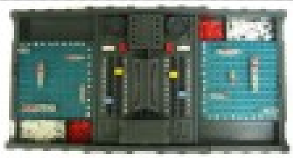
```
#include <array.h>
using namespace std;
E name[]; //declarations
E names[intlength];
E names[7]= {0,1,2,3,4,5,6};
//THERE IS NO SIZE FUNCTION
```

```
/*characters and numbers are always true
according to the ascii table.
0 = false */
```

```
#include <cctype>
using namespace std;
char toupper(Char_Exp)
char tolower(Char_Exp)
bool isupper(Char_Exp)
bool islower(Char_Exp)
bool isalpha(Char_Exp)
bool isdigit(Char_Exp)
//alphanumeric
bool isalnum(Char_Exp)
//!whitespace/letter/digit
bool ispunct(Char_Exp)
```

```
Two dimensional arrays:
string a[rows][columns];
Void f(int a[][N], int n)
//num columns is required!
```

GOOD LUCK!



```
Pointers:
E a;
E* p= &a; //type match!!
void f(E* a) //*=pointer (not
E)
*p = [st] //dereferencing!
f(&a); //reference to variable
```

CS31 Study Guide: operator overloading, dynamic arrays, constructors/destructors, classes, structures, pointers,

Classes

```
class Class_Name
{
    //if unspecified, member is
private
    type_1 member_var_name;
 public:
    Class_Name(); //constructor
    ~Class_Name(); // Destructor
    //accessors
    type_1 getName1();
...
    //mutators
    void   setName1();
...
 private:
    type_3 var_name_3[];
    type_2 var_name_2;
    type 2 function_2();
};   //REMEMBER THIS SEMICOLON!
```

CONSTRUCTORS

```
//if no constructor is specified,
//default constructor is assumed
Class_Name::Class_Name()
{
    member_var_name = default_v;
    var_name_2 = new type_2();
   ...
}
//different constructors allowed
Class_Name::Class_Name(type_1 v)
{
    member_var_name = v;
    var_name_2 = new type_2();
   ...
}
```

DESTRUCTORS

```
/*if no destructor specified,
default is assumed. The
destructor must delete all
dynamically allocated objects,
removing memory leaks*/
Class_Name::~Class_Name()
{
    delete[]  var_name_3;
    delete var_name_2;
/*deleting an array of pointers
to dynamically allocated objects
requires iterating through the
array and deleting each pointer*/
}
```

DON'T PANIC

Structures

```
struct struct_tag
{
    type_1 member_variable_name;
    type_2
member_variable_name_2;
};  //REMEMBER THIS
SEMICOLON!!!!!

struct_tag one; //declaration
// accessing member variables
one.member_variable_name =
value;
```

Const

```
//a won't change
const int a = value;
//v won't change
fct(const int v) {}
//in class, fct won't change class
void fct(int v) const {}
```

Pointers in classes:

```
aStruct c;
c.sPublicVar = value;
sthing = c.getPrivateVar();
astruct* cp = &c;
c.function();
cp->function();
/*this pointer refers to class
instance inside a function from
that class */
```

```
assert(condition);
```

Pointer Arithmetic (in arrays)

```
*&x ➔ x    //pointers and references cancelt sometimes
&a[i] + j = &a[i+j]  //moves down array
&a[i] < &a[j] ➔ i < j //compares order in array
a ⇔ &a[0]   //equivalent
p[i]⇔*(p+i)    //when p is a pointer to a position in an array
&a[i]-&a[j] = i-j //difference in order in array
0 or NULL //null pointer
```

CS31 Study Guide: useful tidbits of code

```
int *p1 = new int[10];
int *p2[15];
for (int i=0; i<15; i++)
    p2[i] = new int[5];
int **p3 = new int*[5];
for (int i=0; i<5; i++)
    p3[i] = new int;
int *p4 = new int;
int *temp = p4;
p4 = p1;
p1 = temp;

//deleting
delete p1;
delete[] p4;
for (int i=0; i<5; i++)
    delete p3[i];
delete p3;
for (int i=0; i<15; i++)
    delete[] p2[i];
```

```
void countMatches
(const char *str1,
const char *str2,
int& count)
{
    count = 0;

    while(*str1 != '\0' &&
    *str2 != '0')
    {
        if(*str1 == *str2)
            count++;
        str1++;
        str2++;
    }
}
```

CS 32 Study Guide: Algorithms, Data Structure vcs, Abstract Data Types, Headers, Linked Lists, Stacks, Queues, Maps, Inheritance

An algorithm is a set of instructions/steps that solve a particular problem.
The imporance of algorithms is: RUNTIME

A data structure the data that's ope ated on by an algorithm to solve a problem.

Abstract Data Type (ADT):
The collection of (a) data structures, (b) algorithms and (c) interface required to solve a particular problem. The ADT provides an interface to secret algorithms and data structures In C++, ADT's are defined as Classes

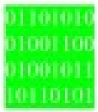
Object Oriented Programming:
programs are co structed from multiple self-contained classes.

Examples of Algorithms:
-Linear search
-Binary search

/* NEVER INCLUDE A .CPP FILE IN ANOTHER FILE. ONLY INCLUDE .H FILES
NEVER PUT 'USING NAMESPACE STD' IN A HEADER*/

Preprocessor Directives:

```
#ifdef FILE_H
 //checks if already defined
#ifndef FILE_H
 //checks if not defined
#define FILE_H
 //defines a constant
#endif //like an end bracket
```



/* use include guards to prevent multiple definitions */

constructors/destructors

/*if you declare an array of objects, that object must have a default constructor that requires no arguments*/

```
Class csNerd
{
  public:
    csNerd(int PCs, bool UsesMac)
      :m_numPCs(PCs), m_MacUser(UsesMac)
      //initializer list
    {…}
    ~csNerd(); //destructor, only one!
}
```

/*desctructors must: Free any dynamically allocated memory, close any opened disk files, and disconnect any opened network connections*/

/* Class co position: If a class contains one or more classes as member variables, */

/*include header files when you define a variable of that class type or call any member function from that class.
DO NOT include header files if you define a parameter, return type or pointer/reference variable of the class */

```
class csNerd;//instead
```

Copying Stuff

```
Class Circ{
  public:
    Circ();
    Circ(const Circ& old);
      //copy constructor
    Circ& operator=(const Circ& source)
      //assignment operator
    {...
      return (*this); //required!
    }
}
int main(){
  circ one;
  circ two;
  two = one; //assignment operator call
  circ three(two); //copy constructor
}
```

/*a default copy constructor performs a shallow copy, which does not work on dynamically allocated data or opened system resources.
A copy constructor must:
- determine how much memory is allocated by the old variable
- allocate the same amount of memory in the new variable
- copy the contents*/

/* the default assignment operator performs a shallow copy, while will not work on dynamically allocated data or any system resources that have been opened.
A assignment operator must:
- free all dynamic memory used by the target instance
- Re-allocate memory in the target instance to hold any member variables from the source instance
- explicitly copy the contents of the source instance to the target instance*/

```
class Stack{
public:
  stack(); //constructor
  void push(int i); //add to stack
  int pop(); //remove from stack
  bool is_empty(void);
  int peek_top(); //return top value
  …
}
```

```
class Queue{
public:
  enqueue(int a); //adds a to end
  int dequeue(); //removes first
  bool isEmpty();
  int size();
  int getFront() //get front value
}
```

Linked Lists: (doubly linked)

```
struct node
{
  string name;
  node* next;
  node* prev;
}

class myLinkedList
{
public:
  void addtoFront(string name);
  void deleteItem(string name);
  void deleteItem(int slotNum);
  int find(string name);
  void print();
  myLinkedList() //creates empty list
  { first = last = NULL }
  ~myLinked List();
private:
  node* first //beg of list
  node* last  //end of list
```

/* You can create linked lists that are singly linked, doubly linked, or in a loop depending on what you need */

CHECK THE BOUNDARY CONDITIONS

/*inert algrithms that insert at the top are the easiest to code and the fastest. Middle/end are slower/more complex*/

/* Destructors must traverse the entire linked list */

DESTRUCTING A DERIVED TYPE

1. Execute the body of the destructor
2. Destroy data members
3. Destroy base part

Linked List Vs. Array

Array is Faster for
- getting a specific item
- less debugging problems

Linked List is Faster for
- inserting at the front
  removing from the middle

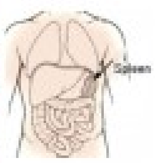
Circular Queue: use pointers head and tail to loop around an array

MAKE SURE THE POINTER DOESN'T POINT TO NULL

CONSTRUCTING A DERIVED TYPE

1. Construct base part
2. Construct data members
3. Execut the body of the constructor

/* Derived classes can only access public member variables and functions of the base class If you want Derived classes, but not the public to access variables, use **protected***/



/* Copy Constructors and assignment operators will copy the base and derived data correctly, UNLESS it is dynamically allo aited */

RECURSION:

1. Identify if the problem is repetitive on a broad scale and/or can be simplified
2. Identify the simplist, complete case
3. Identify the base cases

```
if(base case)
    dosomething
else
    dosomething to reduce the size of
    the problem
```

Inheritance

```
class Base
{
  public
    Base(int p1, int p2)
    void doThis(); //!!!!!
    virtual void doIf(); //default: derived, if it exists
    virtual void doIf2() const =0; //pure virtual
  private:
    [stuff…]
}
class Derived : Public Base
{
  public
    Derived(int p1, int p2) : Base(p1, p2) {}
        //base must be constructed, or default is used
    virtual void doIf2() const;
        //declare overrides virtual as well
    virtual void doIf();
}

void Derived::doIf()
{    Base::doIf2();   }
//to call in a derived class a function from the base
//class that has been overwritten, you need to use
//'Base::'
```

/* Recursive functions should never use global, static, or member variables, only local variables and parameters! */

Generic Programming:

override/define generic comparison operators (<, >, ==, etc)
then, use templates! ☺

TEMPLATE CODE:

```
template <typename T>
   //indicates the following class
   //or function is a template
void function(T a[], T p2)
   //T type must be passed as a
   //parameter!
{
     T total = T(); //see*
     ...
}

void function(int a[], int p2)
{...}  //you can write exceptions the
       //compiler will default to

template <typename T1, T2>
    //multi-type templates work too!
void f2(T1 a[], T2 b[])
```

/* In templates, the compiler uses **template argument deduction** (checks the parameters) to figure out what functions to use. Non-template matches have priority, then te plate matches. If the call does not match the template exactly, there will be a compile time error!*/

/* Using the term T() allows you to initialize to the "default constructor" of whatever type you use. For numbers, this is 0. Bools are false, strings are empty, chars are the 0 byte. */

ALWAYS PLACE TEMPLATES IN THE HEADER FILE

/* when you have a function that traverses the entire leftover list each time, the algorithm has time complexity O(N^2): N(N+1)/2 = 1/2N^2+1/2N)*/

Template Classes

```
template <typename T>
class something
{…};

template <typename T>
void something<T>::f1(T a)
{…};
```

Inline Functions:

/* anything declared inside the class declaration is automatically inline: the compiler copies the code wherever you call the function, speeding up the program because there's less jumping. declare external functions inline like this: */

```
inline void sclass::f1()
{}
```

/* setting large functions inline will greatly increase your exe file size
*/

Runtime Time Complexity

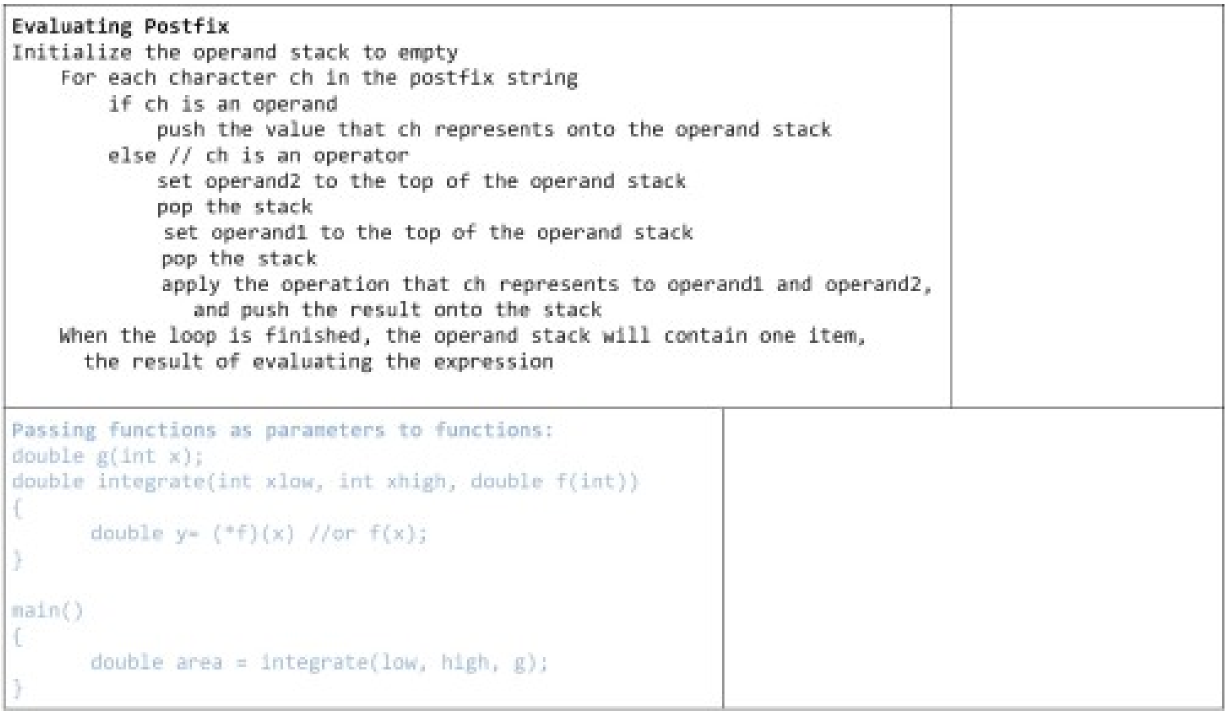
/*written in terms of "Big 'O' Notation" O(some function of N), where N is the number of data terms.

Things to consider if complexity varies:

Best Case Time
Worst Case Time
Average Case Time
Does your data cause you to generate the Best/Worst case often? */

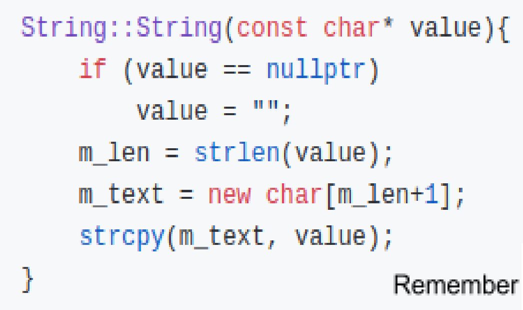
/* sometimes, for things like sorting, you consider complexity of swaps over comparisons (or some other specific action) because it takes significantly longer. Usually, the longer one is not swaps, because you should SWAP POINTERS */

```
INFIX TO POSTFIX
Initialize postfix to null
Initialize the operator stack to empty
For each character ch in the infix string
    Switch (ch)
        case operand:
           append ch to end of postfix
           break
         case '(':
           push ch onto the operator stack
           break
        case ')':
             // pop stack until matching '('
           While stack top is not '('
             append the stack top to postfix
             pop the stack
           pop the stack  // remove the '('
           break
         case operator:
           while the stack is not empty and the stack top is not '('
            and precedence(ch) <= precedence(stack top)
                append the stack top to postfix
                 pop the stack
           push ch onto the stack
           break
While the stack is not empty
    append the stack top to postfix
    pop the stack
```

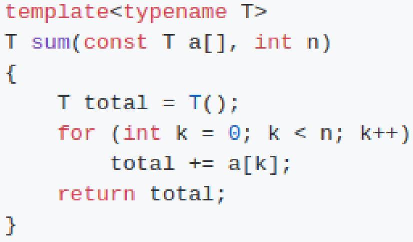


```
Evaluating Postfix
Initialize the operand stack to empty
    For each character ch in the postfix string
        if ch is an operand
            push the value that ch represents onto the operand stack
        else // ch is an operator
            set operand2 to the top of the operand stack
            pop the stack
            set operand1 to the top of the operand stack
            pop the stack
            apply the operation that ch represents to operand1 and operand2,
              and push the result onto the stack
    When the loop is finished, the operand stack will contain one item,
      the result of evaluating the expression
```

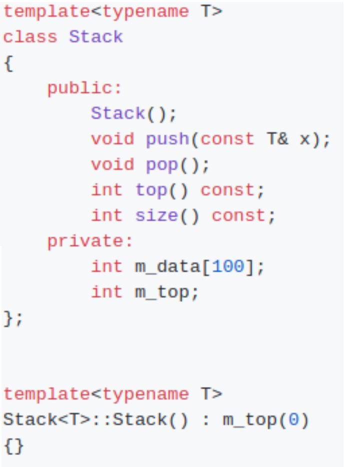
```
Passing functions as parameters to functions:
double g(int x);
double integrate(int xlow, int xhigh, double f(int))
{
        double y= (*f)(x) //or f(x);
}

main()
{
        double area = integrate(low, high, g);
}
```



```
String::String(const char* value){
    if (value == nullptr)
        value = "";
    m_len = strlen(value);
    m_text = new char[m_len+1];
    strcpy(m_text, value);
}
```

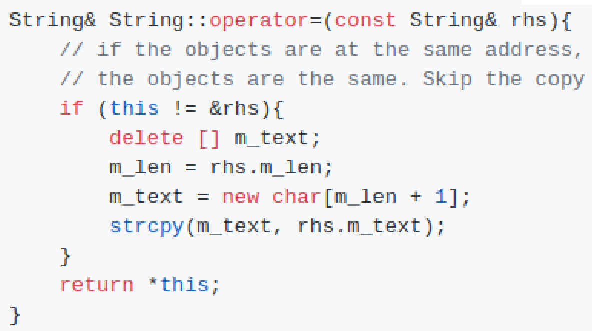


```
template<typename T>
T sum(const T a[], int n)
{
    T total = T();
    for (int k = 0; k < n; k++)
        total += a[k];
    return total;
}
```

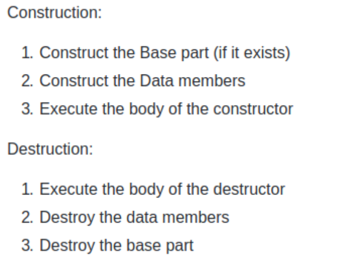


```
template<typename T>
class Stack
{
    public:
        Stack();
        void push(const T& x);
        void pop();
        int top() const;
        int size() const;
    private:
        int m_data[100];
        int m_top;
};

template<typename T>
Stack<T>::Stack() : m_top(0)
{}
```

Remember to check for aliasing issues!



```
String& String::operator=(const String& rhs){
    // if the objects are at the same address,
    // the objects are the same. Skip the copy
    if (this != &rhs){
        delete [] m_text;
        m_len = rhs.m_len;
        m_text = new char[m_len + 1];
        strcpy(m_text, rhs.m_text);
    }
    return *this;
}
```

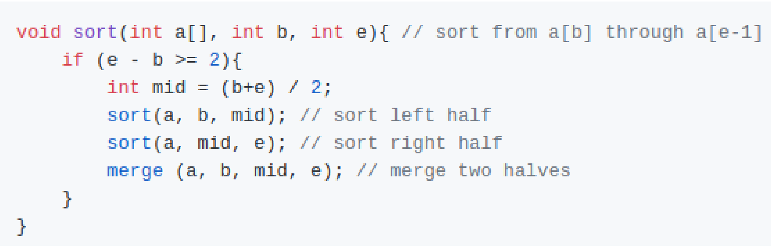


Construction:

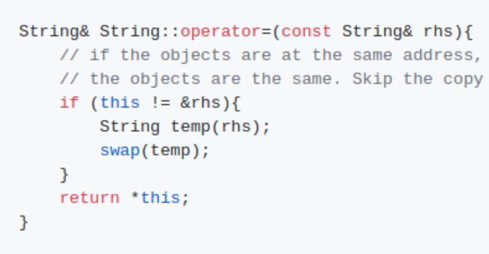
1. Construct the Base part (if it exists)
2. Construct the Data members
3. Execute the body of the constructor

Destruction:

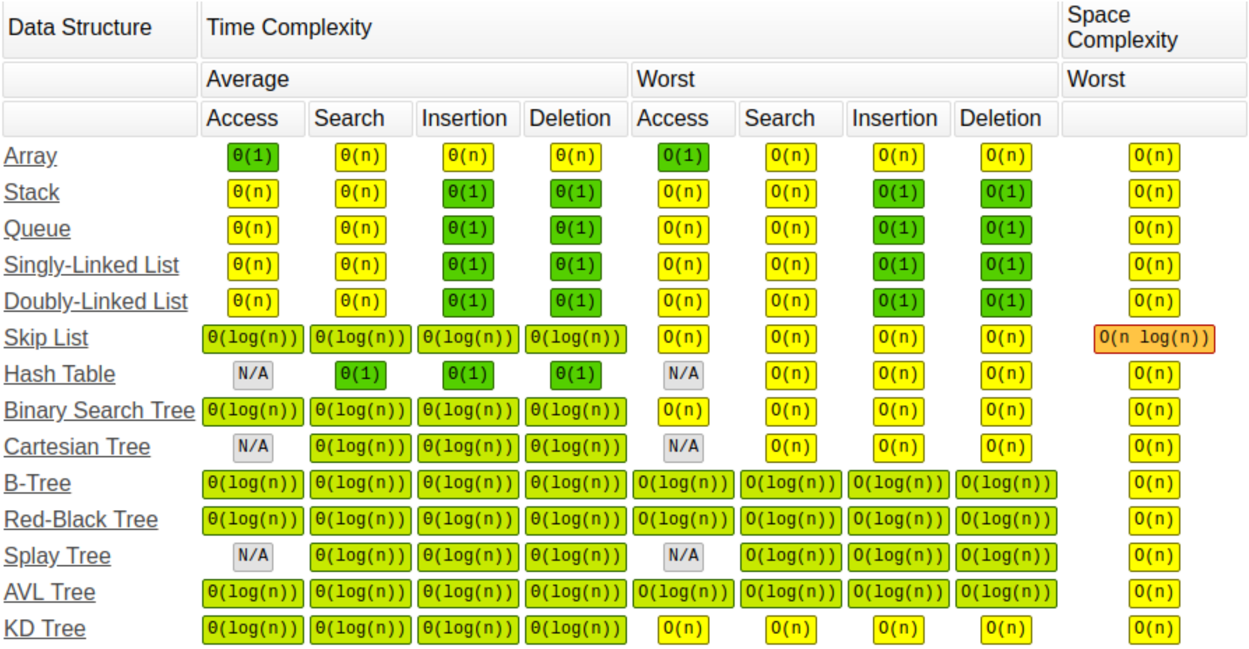
1. Execute the body of the destructor
2. Destroy the data members
3. Destroy the base part



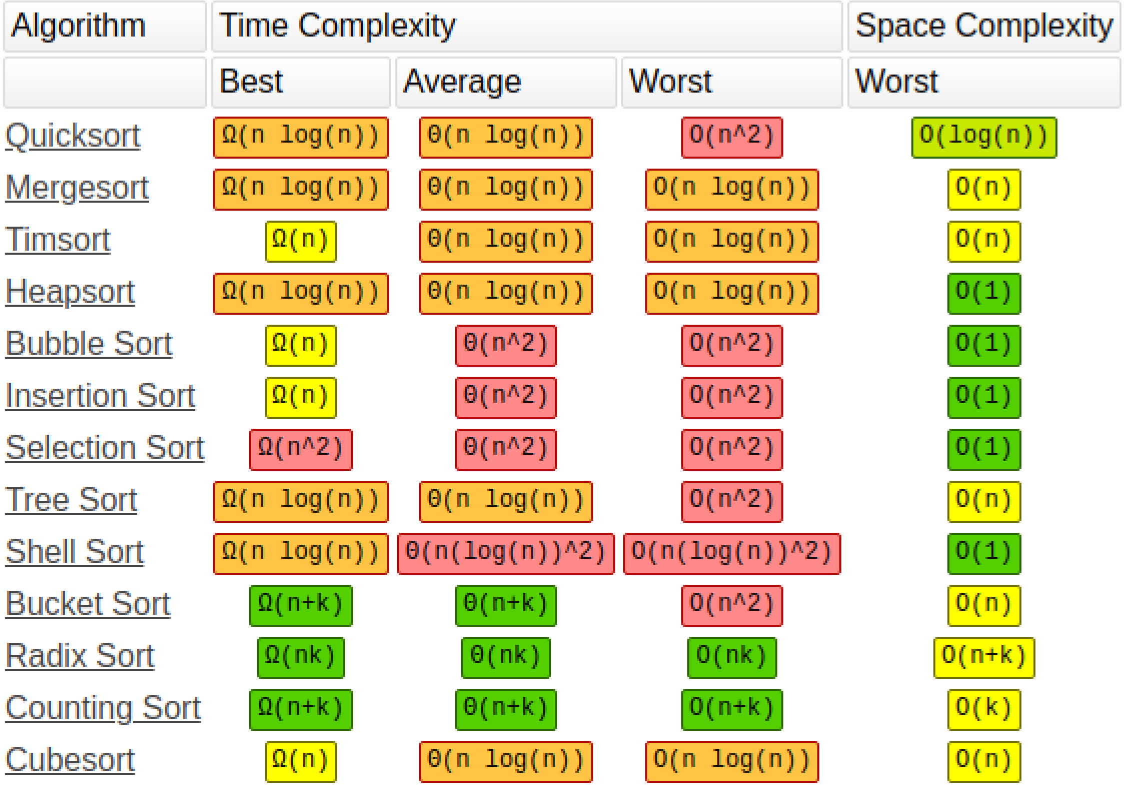
```
void sort(int a[], int b, int e){ // sort from a[b] through a[e-1]
    if (e - b >= 2){
        int mid = (b+e) / 2;
        sort(a, b, mid); // sort left half
        sort(a, mid, e); // sort right half
        merge (a, b, mid, e); // merge two halves
    }
}
```



```
String& String::operator=(const String& rhs){
    // if the objects are at the same address,
    // the objects are the same. Skip the copy
    if (this != &rhs){
        String temp(rhs);
        swap(temp);
    }
    return *this;
}
```



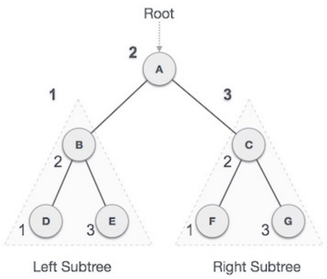
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

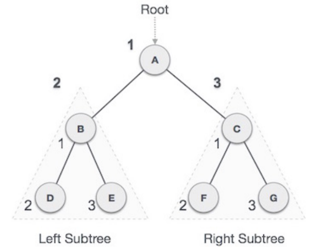
In-order Traversal

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```



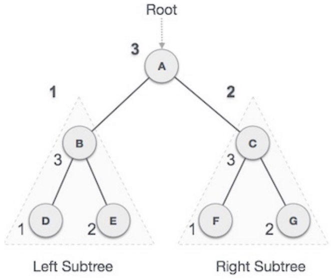
Pre-order Traversal

```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

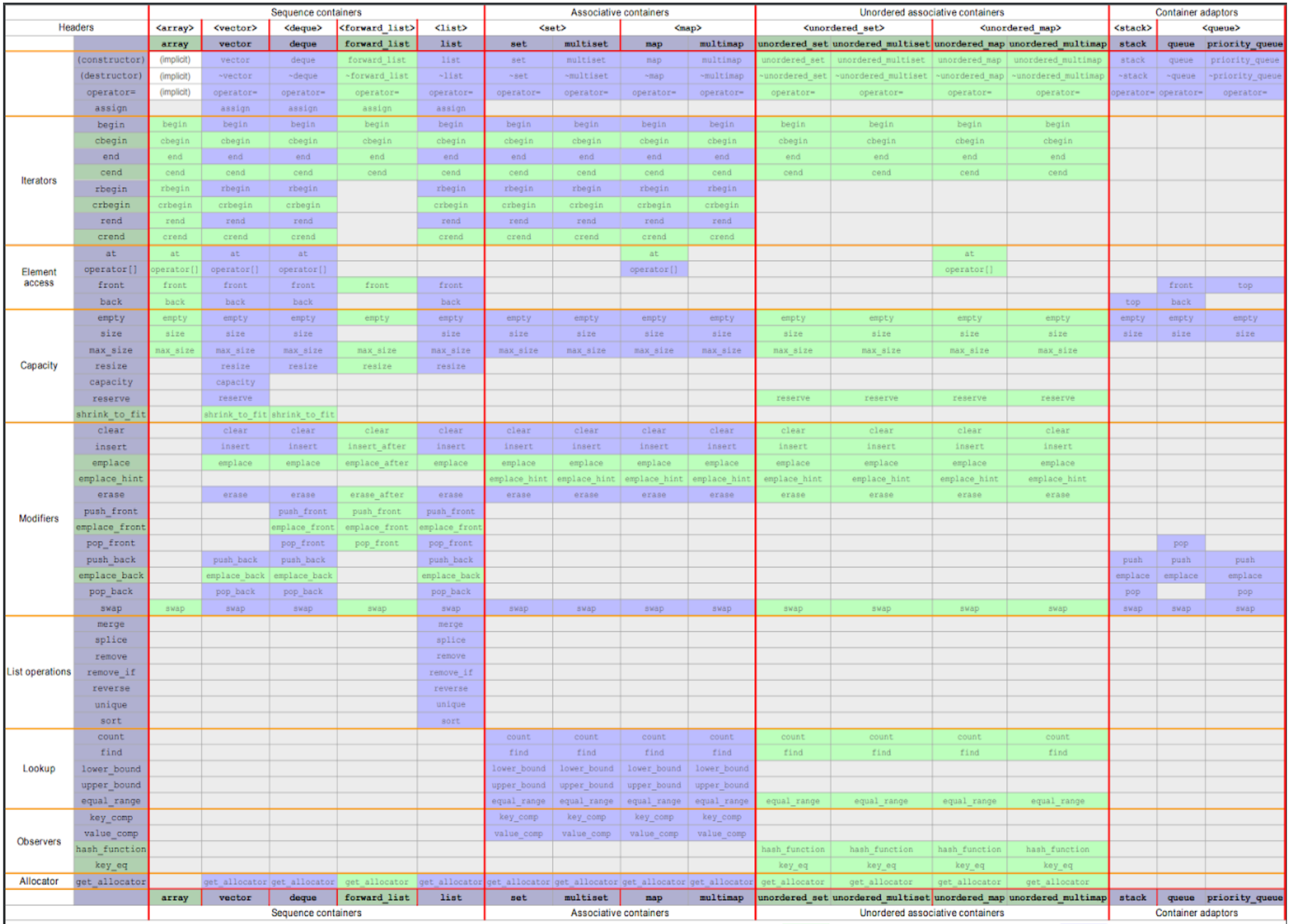


Post-order Traversal

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```



```
//Note how similar this is to rotateLeft
void LinkedList::rotateRight(int n) {
  if (head == nullptr)
    return;

  int size = 1;
  Node* oldTail = head;
  while (oldTail->next != nullptr) {
    size++;
    oldTail = oldTail->next;
  }

  if (n % size > 0) {
    int headPos = size - (n % size);
    Node* newTail = head;
    for (int x = 0; x < headPos - 1; x++) {
      newTail = newTail->next;
    }
    Node* newHead = newTail->next;

    newTail->next = nullptr;
    oldTail->next = head;
    head = newHead;
  }
}
```



		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		
Headers		<array>	<vector>	<deque>	<forward_list>	<list>	<set>		<map>		<unordered_set>		<unordered_map>		<stack>	<queue>	
		array	**vector**	**deque**	**forward_list**	**list**	**set**	**multiset**	**map**	**multimap**	**unordered_set**	**unordered_multiset**	**unordered_map**	**unordered_multimap**	**stack**	**queue**	**priority_queue**
	(constructor)	(implicit)	vector	deque	forward_list	list	set	multiset	map	multimap	unordered_set	unordered_multiset	unordered_map	unordered_multimap	stack	queue	priority_queue
	(destructor)	(implicit)	~vector	~deque	~forward_list	~list	~set	~multiset	~map	~multimap	~unordered_set	~unordered_multiset	~unordered_map	~unordered_multimap	~stack	~queue	~priority_queue
	operator=	(implicit)	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=	operator=
	assign		assign	assign	assign	assign											
Iterators	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin	begin			
	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin	cbegin			
	end	end	end	end	end	end	end	end	end	end	end	end	end	end			
	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend	cend			
	rbegin	rbegin	rbegin	rbegin		rbegin	rbegin	rbegin	rbegin	rbegin							
	crbegin	crbegin	crbegin	crbegin		crbegin	crbegin	crbegin	crbegin	crbegin							
	rend	rend	rend	rend		rend	rend	rend	rend	rend							
	crend	crend	crend	crend		crend	crend	crend	crend	crend							
Element access	at	at	at	at					at				at				
	operator[]	operator[]	operator[]	operator[]					operator[]				operator[]				
	front	front	front	front	front	front										front	top
	back	back	back	back		back									top	back	
Capacity	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty	empty
	size	size	size	size		size	size	size	size	size	size	size	size	size	size	size	size
	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size	max_size			
	resize		resize	resize	resize	resize											
	capacity		capacity														
	reserve		reserve								reserve	reserve	reserve	reserve			
	shrink_to_fit		shrink_to_fit	shrink_to_fit													
Modifiers	clear		clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear	clear			
	insert		insert	insert	insert_after	insert	insert	insert	insert	insert	insert	insert	insert	insert			
	emplace		emplace	emplace	emplace_after	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace	emplace			
	emplace_hint						emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint	emplace_hint			
	erase		erase	erase	erase_after	erase	erase	erase	erase	erase	erase	erase	erase	erase			
	push_front			push_front	push_front	push_front											
	emplace_front			emplace_front	emplace_front	emplace_front											
	pop_front			pop_front	pop_front	pop_front										pop	
	push_back		push_back	push_back		push_back									push	push	push
	emplace_back		emplace_back	emplace_back		emplace_back									emplace	emplace	emplace
	pop_back		pop_back	pop_back		pop_back									pop		pop
	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap	swap
List operations	merge					merge											
	splice					splice											
	remove					remove											
	remove_if					remove_if											
	reverse					reverse											
	unique					unique											
	sort					sort											
Lookup	count						count	count	count	count	count	count	count	count			
	find						find	find	find	find	find	find	find	find			
	lower_bound						lower_bound	lower_bound	lower_bound	lower_bound							
	upper_bound						upper_bound	upper_bound	upper_bound	upper_bound							
	equal_range						equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range	equal_range			
Observers	key_comp						key_comp	key_comp	key_comp	key_comp							
	value_comp						value_comp	value_comp	value_comp	value_comp							
	hash_function										hash_function	hash_function	hash_function	hash_function			
	key_eq										key_eq	key_eq	key_eq	key_eq			
Allocator	get_allocator		get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator	get_allocator			
		array	**vector**	**deque**	**forward_list**	**list**	**set**	**multiset**	**map**	**multimap**	**unordered_set**	**unordered_multiset**	**unordered_map**	**unordered_multimap**	**stack**	**queue**	**priority_queue**
		Sequence containers					Associative containers				Unordered associative containers				Container adaptors		

C++ Containers Library cross-reference table from http://en.cppreference.com/w/cpp/container PDF version with red & orange lines by Robin Whittle December 2012.

🟩 - functions present since C++11

🟪 - functions present in C++03