CSc 300 – Data Structures – Exam #1 – Fall 2020

Gamradt

Name: Nathan Aamodt

1. **(6 points)** State the definition of recursion. A recursive algorithm consists of two types of cases. State what these two case types are called.

Direct Recursion and Indirect Recursion

1. **(4 points)** There are two types of recursion: Direct and Indirect. Briefly describe each one.

Direct – Calls itself

Indirect – calls a different function that calls the recursion function again

1. **(4 points)** State two advantages of using a linked list in place of an array.

Arrays have a set size while linked list can grow and shrink

You can insert a node anywhere in the linked list like in-between nodes

1. **(8 points)** State the three actions that automatically occur when instantiating a class instance in C++ as discussed in class. State when constant members are enforced in relation to these three actions.
2. Storage Allocation – allocated memory can be done dynamically
3. Base Initialization list – const members are locked in at this point
4. Assign using body of constructor – constants can’t be assigned
5. **(4 points)** State the two basic operations used to support a Queue ADT. State what potential problems that need to be considered with each basic operation.

Enqueue

Dequeue

These operations can cause under and overflow issues in your program

1. **(8 points)** Given the following recursive algorithm definition, write the complete function to implement it:  
     
   gcd(x, y) = y; if y divides x evenly  
   gcd(y, remainder of x / y) otherwise

int gcd(int x, y)

{

if(x % y = 0)

return 0;

else

{

y =x%y

gcd(x,y)

}

1. **(8 points)** Define a complete C++ set of user-defined data types to support an integer based linked-list ADT named MyClass. Use the names: Element, Node, NodePtr

typedef int Element;

class MyClass {

public:

MyClass();

MyClass(const MyClass&);

~MyClass();

void MyClass(const Element);

void MyClass(const Element);

void view() const;

void viewR() const;

private:

struct MyClassNode;

typedef MyClassNode \*MyClassNodePtr;

struct MyClassNode {

Element element;

MyClassNodePtr next;

};

MyClassNodePtr head;

void viewR(const MyClassNodePtr) const;

};

1. **(7 points)** Define a complete C++ destructor function for the MyClass class.

#include “MyClass.h”

#include <iostream>

#include<stdio.h>

#include<cstring>

Using namespace std;

Void MyClass :: ~MyClass(const Element num)

{

MyClassNodePtr nodePtr = NULL;

while(head){

nodePtr = head;

head = nodePtr->next;

nodePtr->next = NULL;

delete nodePtr;

}

nodePtr = NULL;

}

1. **(11 points)** Define a complete C++ parameterized constructor function used to create a new empty Stack class instance. The Stack class uses a dynamic array implementation. The desired stack size is passed in at instance creation time. Assume all other Stack class members are available. Assume STACK\_SIZE is a constant member used to support the array. Assume Element is a floating-point data type. Apply reuse of other Stack class members everywhere possible.

typedef double Element;

class Stack {

public:

Stack();

Stack( const int);

Stack( Stack &);

~Stack();

void push(const Element);

Element pop();

Element peek();

void view();

private:

const int STACK\_SIZE;

Element \* stackArray;

int top;

};

1. **(9 points)** Define a complete C++ function named push used to insert one new element to the top of a Stack class instance. The Stack class uses a dynamic array implementation. Assume all other Stack class members are available. Assume STACK\_SIZE is a constant member used to support the array. Assume Element is a floating-point data type. Apply reuse of other Stack class members everywhere possible.

#include “header.h” // not sure what the header would be named

#include <iostream>

#include<stdio.h>

#include<cstring>

Using namespace std;

void Stack::push(const Element e)

{

if (top+1 >= STACK\_SIZE)

{

cout << "Overflow. Stack full please pop or view array."<<endl;

}

else

{

stackArray[top] = e;

top++;

}

}

1. **(16 points)** Define a complete C++ function named dequeue used to remove one existing element from the front of a Queue class instance. The Queue class uses a linked implementation. Assume all other Queue class members are available. Assume Element is a C-style string data type. Assume that exception handling is not supported. Apply reuse of other Queue class members everywhere possible.

#include “header.h” // not sure what the header would be named

#include <iostream>

#include<stdio.h>

#include<cstring>

Using namespace std;

void Queue::dequeue(Element e)

{

QNodePtr nodePtr = NULL;

if (!front)

{

cerr << "Memory not Allocated - No Delete" << endl;

}

else

{

nodePtr = front;

front = front->next;

nodePtr->next = NULL;

strcpy(e, nodePtr->element);

delete nodePtr;

}

nodePtr = NULL;

}

1. **(17 points)** Define a complete C++ function named view used to non-destructively display the contents of an existing Queue class instance. The Queue class uses a linked implementation. Assume all other Queue class members are available. Assume Element is a C-style string data type. Apply reuse of other Queue class members everywhere possible.

FRONT -> BACK // Empty queue FRONT -> CSc -> 300 -> BACK // Populated queue

#include “header.h” // not sure what the header would be named

#include <iostream>

#include<stdio.h>

#include<cstring>

Using namespace std;

void Queue::view()

{

QNodePtr nodePtr = NULL;

nodePtr = front;

cout << "Front->";

while(nodePtr)

{

cout << "->"<<nodePtr->element << endl;

nodePtr = nodePtr->next;

}

cout << "Back" << endl;

nodePtr = NULL;

}