CpSc 2120

Exam 1 Sample Questions

1. Given an IntArray class containing the following data members:

class IntArray

{

private: int \* array; //pointer to array

int size; //size variable

public: //function members

...

};

1. Suppose IntArray contains a function member with the following prototype:

IntArray& operator=(const IntArray & rhs);

This function member is the assignment operator function. Fill in the function definition so that a “deep copy” will be assigned. **(8 pts.)**

IntArray& IntArray::operator=(const IntArray & rhs)

{

if (this != &rhs) //avoid self-assignment

{

size=rhs.size;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

for (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) {

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

}

return \*this;

}

1. Since IntArray contains dynamically allocated data, three function members (ignoring C++11 temporary objects) will need to be implemented by the programmer. One of these is the assignment operator function. What other two functions must be implemented for IntArray? **(4 pts.)**
2. Suppose the IntArray class can perform the following operations. Circle the operator function(s) that **must** be implemented as a **friend** function as opposed to a member function: **(4 pts.)**

operator+ : concatenate two arrays

operator<< : write values to output stream

operator>> : clear array and read comma separated values from input stream

operator== : compare if equal

operator[] : access element in array

1. Why must these operators be implemented as a **friend** function as opposed to a member function **(2 pts.)**
2. Given a linked list class with the functions **insertAtStart()**, **removeFromStart()**, and **removeFromEnd()** that will be used to implement both a **stack** and a **queue**, indicate which linked list function will be used the following operations: **(1 pt. each)**

stack – push \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

stack – pop \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

queue – enqueue \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

queue – dequeue \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Given an ArrayStack class implementing a stack using an array:

class ArrayStack

{

private: int array[10]; //capacity is 10

int top; //indexes item at top (most recently pushed)

public: ArrayStack() : array {0}, top(-1) {}

void push(int i);

int pop();

...};

1. Fill in the function definition for void push(int i); **(5 pts.)**

void ArrayStack::push(int i)

{

if (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

{

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

}

1. Fill in the function definition for int pop(); **(5 pts.)**

int ArrayStack::pop()

{

int val = -1;

if (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

{

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

}

return val; }

1. Given the following CircularLinkedList class that maintains a pointer to a **head** node containing a **next** pointer to the first value-containing node with the first value at position 0 and singly-linked:

class CircularLinkedList | struct Node //all members are public

{ | {

private: Node \* head; | int value;

public: //function members | Node \* next;

}; | };

Fill in the function definition for **find** which will return the position of the first occurrence of **target** in the list (if found) or -1 (if not found): **(10 pts.)**

**head** (no value)

position 0

position 1

position 2

CircularLinkedList with 3 values:

int CircularLinkedList::find(int target)

{

Node \* curNode = \_\_\_\_\_\_\_\_\_\_\_\_;

int pos = \_\_\_\_\_\_\_\_;

while (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

{

curNode = curNode->next;

if (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)

{

return \_\_\_\_\_\_\_\_\_\_\_\_;

}

\_\_\_\_\_\_\_\_\_\_\_\_;

}

return \_\_\_\_\_\_\_\_\_\_\_\_;

}

1. For each of the following Θ, big-O, or big-Ω statements indicate the listed expressions that are represented (list represented expressions by corresponding letter): **(2 pts. each)**
2. Θ(n\*log(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| A: 2n + 5 | B: 2n\*log(n) | C: log(n)/2 + 10 |
| D: n2 + 2n + 5 | E: 50 | F: n\* |
| G: n3/10 + 1 | H: n! | I: 2n |

1. Θ(n2) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. O(n2) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. Ω(n2) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. Θ(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
5. O(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
6. For each fragment of code determine the order of running time and justify your answer. **(3 pts. each)**
7. for (int i=0; i<n; i++)

{

for (int j=n; j>=1; j=j/2)

{

cout << i\*j << endl;

}

}

1. for (int i=0; i<n; i++)

{

cout << i << endl;

}

for (int j=0; j<n; j++)

{

cout << j << endl;

}

1. **Matching:** Give the letter of the corresponding sorting algorithm that best matches the description. **(2 pts. each)**

A. Selection Sort E. Merge Sort H. Bucket Sort

B. Bubble Sort F. Quicksort I. Counting Sort

C. Insertion Sort G. Heapsort J. Radix Sort

D. Shell Sort

1. Randomly pick pivot, all < pivot goes to left of pivot, all > pivot to right of pivot, then recurse
2. Left of current position is sorted, current item walks back until in proper place
3. Sorts values by sorting according to each digit of the value in order starting from either least significant or most significant digit
4. Left to right sweep comparing adjacent values, if backwards swap. After n sweeps, right most n items sorted.
5. Divide in half (recurse until small), sort by combining sorted halves
6. Left of current position is sorted, linear search for next smallest
7. Assign each value to a group with each group only containing values within a certain range then sort the values in each of these groups. Afterwards collect each group of values in order.
8. Iterated insertion sort with decreasing gap size "g" (increment to skip by when inserting into sorted sublist), after iteration with gap size "g" there are "g" interleaved sorted sublists
9. **Multiple Choice: (2 pts. each)**
10. Average case time complexity of Quicksort:

A: Θ(log(n)) B: Θ(n) C: Θ(n\*log(n)) D: Θ(n2)

1. Worst case time complexity of Quicksort:

A: O(log(n)) B: O(n) C: O(n\*log(n)) D: O(n2)

1. Best case time complexity of Shell Sort

A: Ω(n) B: Ω(n\*log(n)) C: Ω(n\*) D: Ω(n\*)

1. Space complexity of Merge Sort:

A: O(1) B: O(log(n)) C: O(n) D: O(n\*log(n))

1. Space complexity of Heapsort:

A: O(1) B: O(log(n)) C: O(n) D: O(n\*log(n))

1. Which sorting algorithm is adaptive?

A: Selection Sort B: Insertion Sort C: Merge Sort D: Heapsort

1. Which of the following is a non-comparison sort?

A: Quicksort B: Heapsort C: Radix Sort D: Shell Sort

1. A sorting algorithm is used to sort a list of city names along with their state/province in alphabetical order according to city name. The original unsorted list and the sorted list are shown below:

Unsorted: Sorted:

Montreal, QC Albuquerque, NM

Portland, OR Greenville, NC

Albuquerque, NM Greenville, SC

Vancouver, BC Montreal, QC

Greenville, SC Portland, OR

Portland, ME Portland, ME

Greenville, NC Vancouver, WA

Vancouver, WA Vancouver, BC

Based on the sorted list the sorting algorithm used is: A: Stable B: Unstable

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 3 | 6 | 4 | 7 | 1 | 8 | 2 |

1. Given the following array draw out how Merge Sort will divide the array into two sublists at each recursion level down to the base case and then how these sublists are merged back into the sorted array: **(8 pts.)**

Extra Credit: **(5 pts.)**

Given the following infix expression draw out the steps taken to convert the infix expression to a postfix expression using a stack:

A \* B + (C – D \* E) \* F

|  |
| --- |
|  |
|  |
|  |
| stack |

|  |
| --- |
| A |
| output |

Read A:

|  |
| --- |
|  |
|  |
| \* |
| stack |

Read \*:

|  |
| --- |
| A |
| output |

|  |
| --- |
|  |
|  |
| \* |
| stack |

Read B:

|  |
| --- |
| A B |
| output |