ATILIM UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

**Name: Name: Id:**

**Section:**

**Signature:**

**CMPE 226 Data Structures**

**Year** : 2018-2019 Fall

**Instructors** : E. Gökçay

**Mid-2** Examination

**Date**: 29.11.2018 **Time**: 16:30-17:55

**Duration**: 85 minutes

**WARNINGS**

* It is forbidden to bring electronic data storage equipments (mobile phones, MP3 players, flash disks and so on.) to exams.
* Students who either cheat, attempt to cheat or provide a help to other(s) in cheating, get 0 (zero) grade from this examination. Also, based on the regulations, a disciplinary action will be taken.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Q1 (10) | Q2 (15) | Q3 (25) | Q4 (10) | Q5 (15) | Total 100 |
|  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Q1 (20) | Q2 (20) | Q3 (20) | Q4 (15) | Q5 (5) | Q6 (10) | Q7 (10) | Total 100 |
|  |  |  |  |  |  |  |  |

**Q1) (20pts)** Assume that you have a hash table of size 100 and you are using the following hash function using integer (all positive integers) keys. Is this a good hash function? Why/Why not? Consider 3 properties of a hash function. Explain.

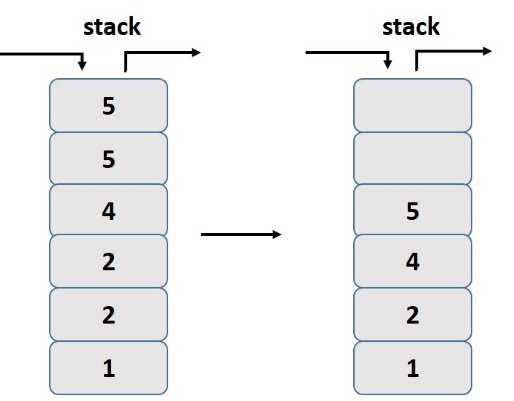
index = (**key** mod 100) / 10;

“key mod 100” will distribute all positive integers between 1..100 and “/ 10” will distribute these keys between 1..10. So with this hash function you can only use %10 percent of the HashTable. So the hash function cannot distribute keys evenly to the hash table. Otherwise it is O(1) and fast algorithm.

NOT A GOOD HASH FUNCTION

**Q2) (20pts)** Assume that you have a stack that contains a set of sorted integer numbers. Write a main program to remove duplicate entries (2 or more) from the stack. **You can use another stack but nothing else.** Define all variables.

HINT: Move all elements from one stack to another and check for duplicates.



Available stack operations:

st.top() : return top data

st.pop() : return and delete top data

st.push(data) : push data to the top of stack

Stack<int> dup; // Assume the stack is full

Stack<int> uniq; // Other empty stack

int main(){

if (dup.isEmpty()) return;

while(!dup.isEmpty()){

uniq.push(dup.pop()); // move one element

// Eliminate duplicates from dup.

while ( !dup.isEmpty() && dup.top() == uniq.top() ) {

dup.pop(); // delete the duplicate from the original array

}

}

// To preserve original order copy back

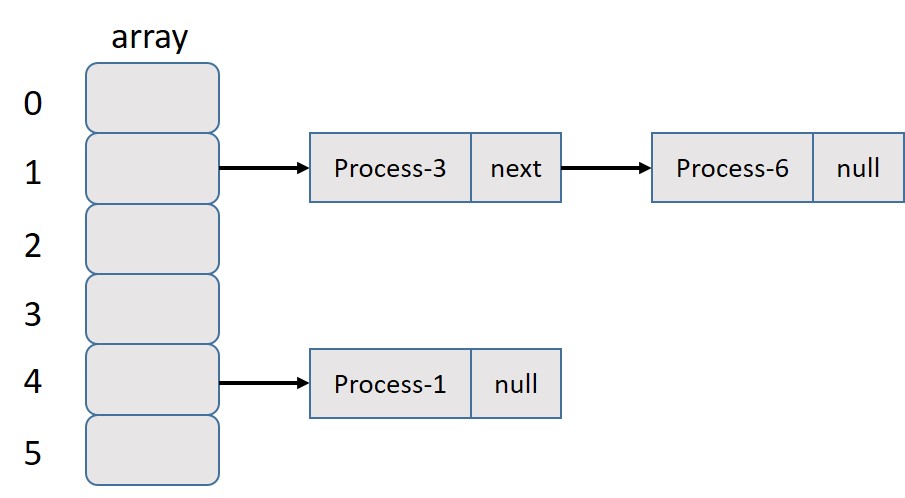
while(!uniq.isEmpty()){

dup.push(uniq.pop());

}

}

**Q3) (20pts)** Assume that you have the following Priority Queue Implementation. The array indexes define the priority level. For a given priority level, the processes (name of the process as a string) are stored using a linked list. Assume that you want to **change** the priority of “Process-6” from level 1 to level 4. Define the variable for the Priority Queue and write a main program for the operation. You can assume that the Priority Queue already contains the data below. Assume LinkedList class and methods are given.



// Array of LinkedList. Assume that the data is loaded.

LinkedList<string> PQ[10];

// You can search all array locations as well

node<string> \*p = PQ[1].searchList("Process-6");

// you can insert as a node or as data

if (p != NULL) PQ[4].insertLast(p->data);

// delete node using the node pointer or data

PQ[1].deleteNode("Process-6");

**Q4) (15pts)** Here is an INCORRECT pseudocode for the algorithm using a stack which is supposed to determine whether a sequence of parentheses is balanced:

declare a character stack

while ( more input is available)

{

read a character

if ( the character is a '(' )

push it on the stack

else if ( the character is a ')' and the stack is not empty )

pop a character off the stack

else **// the character is a ')' and the stack is empty**

print "unbalanced" and exit

}

print "balanced"

Which of these unbalanced sequences does the above code think is balanced? Trace and explain.

1. **( ( ( ) )**  => This choice will print “balanced”. Since # of “)” is less, the stack is not empty with one “(“ left. But the algorithm does not check this condition. After finishing all input characters, it should check if the stack is empty or not.
2. ( ) ) ( ( )
3. ( ( ) ( ) ) )
4. ( ( ) ) ) ( )

**Q5) (5pts)** In the array version of the stack class (with a fixed-sized circular array), which operations require linear time for their worst-case behavior? You can mark one or more choices if necessary.

A) is\_empty

B) top

C) pop

D) push

E) **None of these operations require linear time.** Since it is a circular array, with an index variable pointing to the top, all operations can be implemented in constant time.

**Q6) (10pts)** Calculate the following postfix expression

Postfix Value?

|  |  |
| --- | --- |
| 2 4 5 \* - 3 1 2 + + / | -3 |

**Q7) (10pts)** Convert the following postfix expression to infix (Don’t calculate, just convert). You may use paranthesis in the infix part.

Postfix Infix?

|  |  |
| --- | --- |
| a b + c \* d e + / | (a+b)\*c / (d + e) |