Second part of the final test of Algoritmos e Estruturas de Dados

February 4, 2022 **17h00m – 17h55m**

Nome: _____

N. Mec.: _____

4.0 **1:** Explain how a *min-heap* is organized. For the *min-heap* presented below, insert the number 3. Do not present only the final result, present, step by step, what happens to the array.

1	4	2	6	5	8	7	9	
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

4.0 **2:** Explain how *insertion sort* works. What is its computational complexity? What are its best are worst cases?

4.0 **3:** Explain how you can search for an item of information in a doubly-linked list. What is its computational complexity? What can you do to inprove the serch time of the most frequently serched items?

4.0 **4:** A programmer wants to use a hash table to count the number of distinct words in a text file. She is expecting a text file with about 6000 distinct words, and so she used separate chaining and an array with 10007 entries. She also used the following hash function:

```
unsigned int hash_function(unsigned char *s,unsigned int hash_table_size)
{
  unsigned int sum = 0u;

  for(int i = 0;s[i] != '\0';s++)
    sum += (unsigned int)(i + 1) * (unsigned int)s[i];
  return sum % hash_table_size;
}
```

Unfortunately, her expectations were wrong, and one of the text files has 1000000 distinct words. Answer to the following questions:

1.0 **a)** The given *hash function* is acceptable. Why?

3.0 **b)** With *separate chaining* the *hash table* can accomodate the one million distinct words, even with an array with only 10007 entries. Explain how, and explain by how the performance of this data structure is degraded.

4.0 **5:** The following five functions visit the nodes of the binary tree depicted below in different orders. The number inside each node is the visitation number (so 1 corresponds to the first node for which the visit function was called. Each of the four visitation orders sorrespond to zero or more of the five functions. Identify which ones correspond to each visitation order.

```
void f1(tree_node *n)
{
  queue *q = new_queue();
  enqueue(q,n);
  while(is_empty(q) == 0)
  {
    n = dequeue(q);
    if(n != NULL)
    {
      enqueue(q,n->right);
      enqueue(q,n->left);
      visit(n);
    }
  }
  free_queue(q);
}
```

```
void f2(tree_node *n)
{
  stack *s = new_stack();
  push(s,n);
  while(is_empty(s) == 0)
  {
    n = pop(s);
    if(n != NULL)
    {
      push(s,n->right);
      visit(n);
      push(s,n->left);
    }
  }
  free_stack(s);
}
```

```
int cnt = 0;

void visit(tree_node *n)
{
  printf("%d\n",++cnt);
}
```

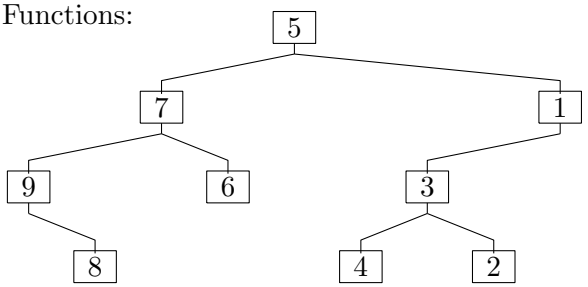
```
void f3(tree_node *n)
{
  if(n != NULL)
  {
    visit(n);
    f3(n->left);
    f3(n->right);
  }
}
```

```
void f4(tree_node *n)
{
  if(n != NULL)
  {
    f4(n->right);
    visit(n);
    f4(n->left);
  }
}
```

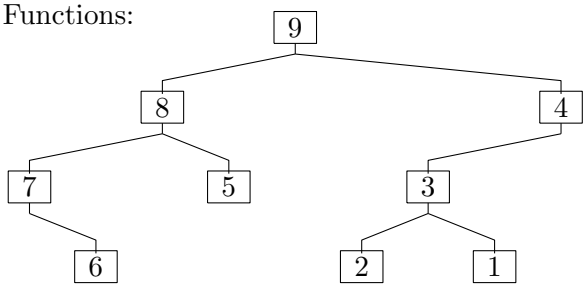
```
void f5(tree_node *n)
{
  if(n != NULL)
  {
    f5(n->right);
    f5(n->left);
    visit(n);
  }
}
```

m

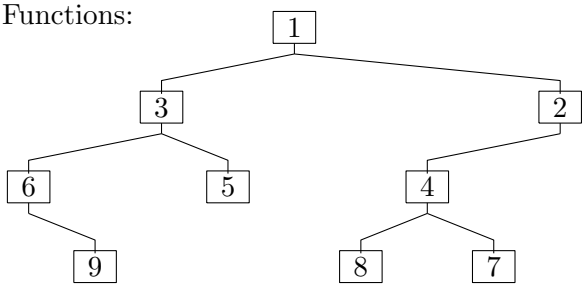
Functions:



Functions:



Functions:



Functions: