PSD

Esame 28 giugno

Il codice va obbligatoriamente indentato (in quanto parte della valutazione). Usare gli spazi per indentare il codice.
Per tutte le domande scrivere il codice e spiegare il funzionamento dell'algoritmo utilizzato (evitare di spiegare ogni riga di codice se non strettamente necessario).

1. Implementare l'ADT **Coda** mediante l'ausilio di due stack (usare l'ADT Stack). Implementare la relativa struttura dati e i seguenti operatori:
   a. newQueue
   b. isEmptyQueue
   c. enqueue
   d. dequeue
2. Si implementi un algoritmo che, dato un albero binario, restituisca, per tutti i nodi con chiave minore ad una data chiave k, una List contenente tali chiavi.
   Prototipo: List funzione(Btree T, Item k)
3. Implementare una funzione reverse queue ricorsiva. La funzione inverte l'ordine degli elementi di una coda (non occorre stampare a video il risultato). Spiegare anche il meccanismo della ricorsione.

1)

```
#define MAX_QUEUE 50

struct queue{
  Stack s1;
  Stack s2;
};

Queue newQueue()
{
  Queue q = malloc(sizeof(struct queue));
  if(!q)
    return NULL;

  q->s1 = newStack();
  q->s2 = newStack();
  return q;
}
```

```
int isEmptyQueue(Queue q)
{
  return isEmptyStack(q->s1);
}

int enqueue(Queue q, Item it)
{
  return push(q->s1, it);
}
```

```
34  Item dequeue(Queue q)
35  {
36    Item it;
37    if(isEmptyQueue(q))
38      return NULL;
39
40    while(!isEmptyStack(q->s1)){
41      it = top(q->s1);
42      pop(q->s1);
43      if(!isEmptyStack(q->s1))
44        push(q->s2,it);
45    }
46    while(!isEmptyStack(q->s2)){
47      Item it2 = top(q->s2);
48      pop(q->s2);
49      push(q->s1,it2);
50    }
51    return it;
52  }
```

```
54  void printQueue(Queue q)
55  {
56    printStack(q->s1);
57  }
```

2)

```
List funzione(BTree bt,Item k){
    List list=newlist();
    Stack stack=newStack();
    push(stack,bt);

    while(!isEmptyStack(stack)){
        BTree temp=pop(stack);
        if(!isEmptyTree(temp->right))
            push(stack,temp->right);
        if(!isEmptyTree(temp->left))
            push(stack,temp->left);
        if(cmpItem(getBTreeRoot(temp),k)<0)
            addListTail(list,getBTreeRoot(temp));
    }
}
```

3)

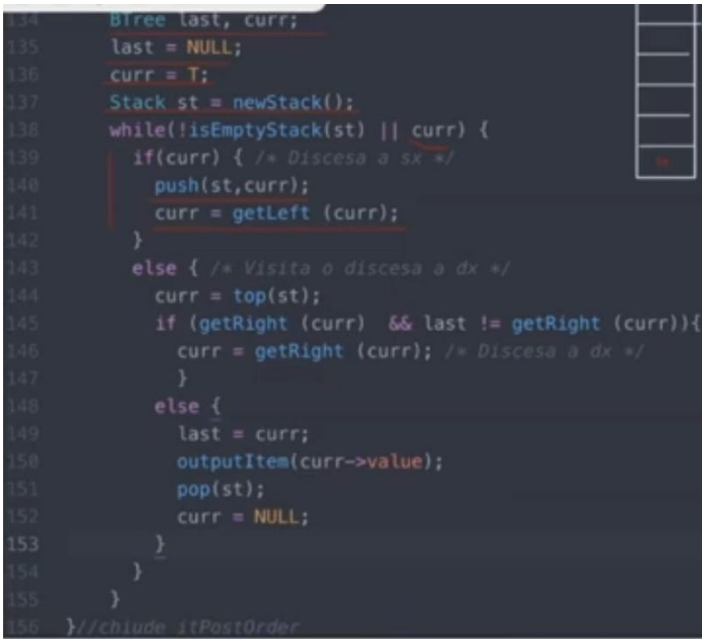
REVERSEQUEUE

Lavoro di combinazione costante:
$T(n) = a_1\ T(n-1)$
Lineare con n

```
6   void reverseQueue(Queue q)
7   {
8     if(isEmptyQueue(q))
9       return;
10    Item it = dequeue(q);
11    reverseQueue(q);
12    enqueue(q,it);
13  }
```

14/07
1)Visita iterativa albero

void itPostOrder(BTree t){



2)inserisci nella lista i valori delle foglie dell’albero

```
117  void byLevel(BTree bt){
118      BTree left, right;
119      Queue q = newQueue();
120      enqueue(q, bt);
121      while (!isEmptyQueue(q)){
122          BTree node = dequeue(q);
123          outputItem(node->value);
124          if ((left = getLeft(node))!=NULL)
125              enqueue(q, left);
126          if ((right = getRight(node))!=NULL)
127              enqueue(q, right);
128      }
129  }
```

3)bilanciamento espressione

```
6   int isOpen(char ch){
7     if (ch=='(' || ch=='[' || ch=='{')
8       return 1;
9     return 0;
10  }
11
12  int isClosed(char ch){
13    if (ch==')' || ch==']' || ch=='}')
14      return 1;
15    return 0;
16  }
```

```
18  int isCorresponding(char ch1, char ch2){
19    if (isOpen(ch1) && isClosed(ch2) && ((ch2-ch1)==1 || (ch2-ch1)==2))
20      return 1;
21    return 0;
22  }
```

```
int isBalanced(char *exp){
  Stack stack = newStack();

  if (*exp=='\0')
    return 1;
  for (; *exp!='\0'; exp++)
  {
    if (isOpen(*exp)){
      Item item = exp;
      outputItem (item);
      push (stack, item);
    }
    else if (isClosed(*exp)){
      if (isEmptyStack(stack))
        return 0;
      else{
        char *x=top(stack);
        pop(stack);
        if (!isCorresponding(*x, *exp))
          return 0;
        }
    }
  }
  return isEmptyStack(stack);
}
```

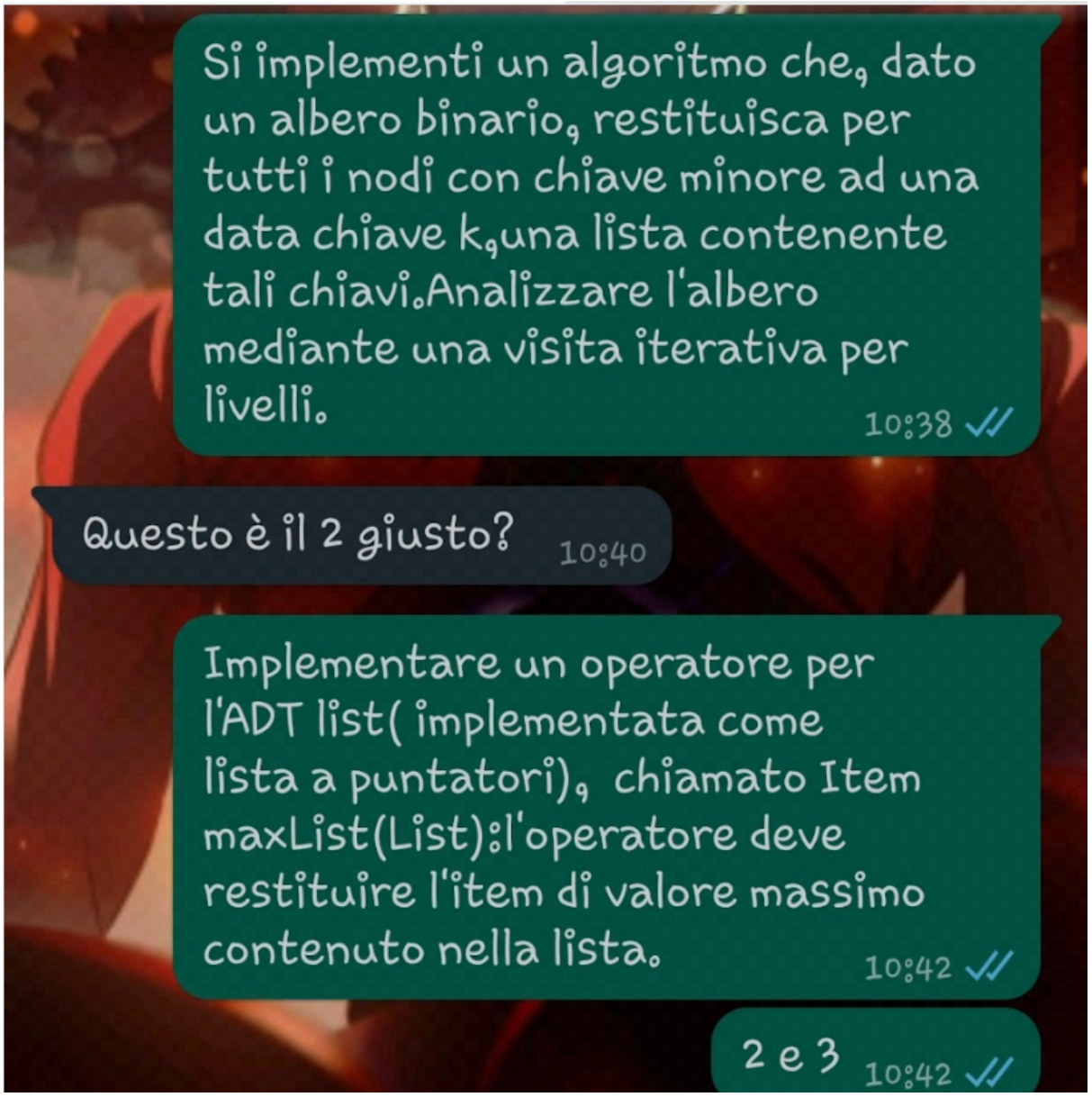
14/09

1)Implementare la reverse queue ricorsiva

REVERSEQUEUE

Lavoro di combinazione costante:
$T(n) = \alpha_1 T(n-1)$
Lineare con n

```
6   void reverseQueue(Queue q).
7   {
8     if(isEmptyQueue(q))
9       return;
10    Item it = dequeue(q);
11    reverseQueue(q);
12    enqueue(q,it);
13  }
```



2)

void itPostOrder(BTree t){

```
BTree last, curr;
last = NULL;
curr = T;
Stack st = newStack();
while(!isEmptyStack(st) || curr) {
  if(curr) { /* Discesa a sx */
    push(st,curr);
    curr = getLeft (curr);
  }
  else { /* Visita o discesa a dx */
    curr = top(st);
    if (getRight (curr)  && last != getRight (curr)){
      curr = getRight (curr); /* Discesa a dx */
      }
    else {
      last = curr;
      outputItem(curr->value);
      pop(st);
      curr = NULL;
    }
  }
}
}//chiude itPostOrder
```

3)Cercare l'item di valore massimo nella lista

CERCARE UN ITEM IN UNA LISTA

```
Item searchNode (struct node *node, Item item, int* pos)
{
  if (node == NULL)
  {
    *pos = -1;
    return NULL;
  }
  if (cmpItem (node -> item, item) == 0)
    return node -> item;
  else
  {
    ++*pos;
    return searchNode (node -> next, item, pos);
  }
}
Item searchListRec (List list, Item item, int* pos)
{
  *pos = 0;
  return searchNode (list -> head, item, pos);
}
```

3/11

Scrivere una funzione che restituisca i valori minimo e massimo contenuti nei nodi di un albero binario (NOTA: l'albero **non** è ordinato). **Descrivere inoltre la complessità computazionale dell'algoritmo.**

```
struct result {
    int min;
    int max;
};
```

```
190  int max3(int a, int b, int c){
191      return a > b ? (c > a ? c : a) : (c > b ? c : b);
192  }//chiude Max3
193  int getMax(BTree T) {
194      if (isEmptyTree(T))
195        return -1;
196      int max_dx, max_sx;
197      int *pt = T->value;
198
199      max_sx=getMax(getLeft(T));
200      max_dx=getMax(getRight(T));
201
202      return max3(max_sx, max_dx, *pt);
203    }//chiude getMax
```

Si implementi una procedura **ricorsiva** che, dato un albero binario, restituisca una lista contenente solo i nodi che abbiano almeno **2** nodi figli. Spiegare **anche** il meccanismo della ricorsione in generale.

Implementare una funzione *sortSongs(Playlist P)* come operatore di ADT playlist. La funzione ordina usando un **selection sort** gli elementi della playlist secondo il titolo della canzone. Le precondizioni e le postcondizioni sono illustrate nell'immagine.

sortSongs(Playlist) → Playlist

sortSongs(pl) → pl'
- Pre: n>0
- Post: pl'.songs = ...
  titolo(a[i]) < titolo(a[i+1])

Implementare tutte le funzioni dell'ADT List e ADT Item interessate.

Utilizzare le seguenti strutture dati senza modificarle.

```
struct playlist{
    char *name;
    List songs;
};

struct list {
    int size;
    struct node *head;
};

struct song{
    char *title;
    char *artist;
    int duration;
};
```

SELECTION_SORT_RIC

```
36  void selection_sort_ric(Item a[], int n) {
37    if(n==1)
38      return;
39
40    int min = minimo(a+1, n-1)+1;
41
42    if (cmpItem(a[min],a[0])<0)
43      swap(&a[0], &a[min]);
44
45    selection_sort_ric(a+1, n-1);
46  }
```

Lavoro di combinazione lineare
a) $T(n) = T(n-h) + b\,n + d$
Quadratico con n

MINIMO

```
15  int minimo(Item *a, int n) {
16      int min = 0, i;
17
18      for(i=1; i<n; i++)
19          if (cmpItem(a[i],a[min])<0)
20              min=i;
21
22      return min;
23  }
```