1. **Punti 2** – Indicare quali delle seguenti affermazioni sulle macro sono vere:
   * 1. Una macro può essere vuota
     2. Le macro non possono essere presenti negli header file
     3. Le macro vengono risolte nella fase di linking
     4. Le macro vengono risolte attraverso una sostituzione lessicale
2. **Punti 2** – Nella compilazione separata di un progetto composto da più file sorgenti (sono possibili 0 o più risposte):
   * 1. Ogni file sorgente dà origine ad un file oggetto
     2. Un file oggetto è un programma completo
     3. Il linking con le librerie di sistema è un linking statico
     4. Il compilatore segnala errori di visibilità
3. **Punti 4** – Domanda a risposta aperta  
   Cosa stampa il seguente programma: \_\_\_\_\_21 1 10 4\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**#include** <iostream>

**using** **namespace** std ;

**int** **f**(**int** i, **int** \*p, **int** &ri, **int** \*&rp) {

i = 10 ;

\*p = i++ ;

ri = \*p\*2;

rp = p+1;

(\*rp)++;

**return** i+\*p;

}

**int** **main**() {

**int** a[] = {1,2,3}, b=6;

**int** \*prp = &a[0];

cout<<f(\*a, a+1, b, prp)<<" ";

**for**(**int** i=0;i<3;i++)

cout<<a[i]<<" ";

}

1. **Punti 3** – Domanda a risposta aperta  
   Cosa stampa il seguente programma: \_\_\_\_\_ 3 2 1 1 2 3\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**#include** <iostream>

**using** **namespace** std ;

**void** **printFun**(**int** test)

{

**if** (test < 1)

**return**;

**else**

{

cout << test << " ";

printFun(test-1);

cout << test << " ";

**return**;

}

}

**int** **main**()

{

**int** test = 3;

printFun(test);

}

1. **Punti 8** – Scrittura di codice

Date le seguenti dichiarazioni per un binary search tree con chiave intera e ordinata in *ordine crescente*:

typedef int tipo\_key;

struct bnode {

int key;

bnode\* left;

bnode\* right;

bnode\* parent;

};

1. **Punti 4** – Scrittura di codice

Scrivere il codice della primitiva per l’inserimento di un nodo **bst\_insert** (è possibile usare altre primitive nella implementazione della primitiva):

void bst\_insert(bst& b, bnode\* n){

bnode\* x;

bnode\* y=NULL;

if(b==NULL){

b=n;}

else{

x=b;

while (x != NULL) {

y=x;

if (get\_key(n)<get\_key(x)) {

x = get\_left(x);}

else {

x = get\_right(x);

}

}

n->parent = y;

if (get\_key(n)<get\_key(y)) {

y->left = n;

} else {

y->right = n;

}

}}

1. **Punti 4** – Scrittura di codice

Scrivere la funzione void print\_subtree(bst b, int x) che dato un valore intero x presente nel BST b, stampi in *ordine decrescente* tutti i valori nel sottoalbero radicato nel nodo che contiene x (nell’implementazione è possibile scrivere più funzioni):

void print\_subtree(bst b, int x){

bool found = false;

while (b != NULL&&!found) {

if (k==get\_key(b))

found = true;

if (compare\_key(k,get\_key(b))<0) {

b = get\_left(b);

} else {

b = get\_right(b);

}

}

print\_BST(b);

}

void print\_BST(bst b){

if(get\_left(b)!=NULL)

print\_BST(get\_right(b));

print\_key(get\_key(b));

cout<<" ";

print(get\_value(b));

cout<<endl;

if(get\_right(b)!=NULL)

print\_BST(get\_left(b));

}

1. **Punti 4** – Scrittura di codice

Date la seguente dichiarazione di coda e primitive su tipo\_inf, scrivere il codice della primitiva enqueue:

struct elem

{

tipo\_inf inf;

elem\* pun ;};

typedef elem\* lista;

typedef struct{

lista head;

elem\* tail;} coda;

int compare(tipo\_inf,tipo\_inf);

void copy(tipo\_inf&,tipo\_inf);

void print(tipo\_inf);

coda enqueue(coda c, tipo\_inf i){

elem\* p = new elem ;

copy(p->inf,inf);

p->pun=NULL;

if(c.tail!=NULL)

c.tail->pun=e;

c.tail=e;

if(c.head==NULL)

c.head=c.tail;

return c;

}

1. **Punti 4** – Scrittura di codice

Data una lista *l* semplice di interi, scrivere la funzione **delete\_even** che cancella da *l* tutti gli elementi in posizione pari (2°,4°,…). Ad esempio per la lista [1,3,2,5,6] il risultato deve essere [1,2,6]. La funzione deve far uso esclusivamente delle primitive specificate sotto:

**struct** elem

{

**int** v;

elem\* pun ;

} ;

**typedef** elem\* lista ;

tipo\_inf head(lista);

lista tail(lista);

void **delete\_even**(lista& l){

lista app = l;

while (app!=NULL){

elem\* del = app->pun;

if(app->pun!=NULL)

app->pun=app->pun->pun;

delete del;

app = app->pun;

}

}

1. **Punti 4** – Scrittura di codice

Date le seguenti dichiarazioni di albero e relative primitive, scrivere la funzione booleana path(node\*e) che restituisca true se esiste un cammino dal nodo e ad una foglia che contenga solo nodi con lo stesso valore di e, false altrimenti.

**struct** node {

int inf;

node\* firstChild;

node\* nextSibling;

};

**typedef** node\* tree;

int **get\_info**(node\*); //restituisce il contenuto informativo del nodo

node\* **get\_firstChild**(node\*); //resituisce il primo figlio del nodo

node\* **get\_nextSibling**(node\*); //restitusce il prossimo fratello del nodo

**bool** **path**(node\* n){

tree t1 = get\_firstChild(n);

if (t1==NULL)

return true;

**bool** ris = **false**;

**while**(t1!=NULL&&!ris){

**if(get\_info(n)==get\_info(t1))**

ris = path(t1);

**if** (!ris&& **get\_info(n)==get\_info(t1)**)

t1 = get\_nextSibling(t1);

}

**return** ris;

**}**

1. **Punti 5** – Scrittura di codice

Data la seguente definizione di grafo, scrivere la funzione booleana same\_degree che restituisca true se tutti i nodi hanno lo stesso numero di archi uscenti, false altrimenti:

**struct** adj\_node {

**int** node;

**float** weight;

**struct** adj\_node\* next;

};

**typedef** adj\_node\* adj\_list;

**typedef** **struct**

{adj\_list\* nodes;

**int** dim;

} graph;

bool same\_degree(graph g){

int i;

int \* v=new int[get\_dim(g)]; /\*vettore che contiene il grado di ogni nodo \*/

for(i=0; i<get\_dim(g);i++)

v[i]=0;

for(i=1;i<=get\_dim(g);i++){

adj\_list l=get\_adjlist(g,i);

while(l!=NULL){

v[i-1]++;

l=get\_nextadj(l);

}

}

int deg=v[0];

for(i=1;i<get\_dim(g);i++)

if(v[i]!=deg)

return false;

return true;

}