Liste
Definizione e Primitive

```
list  emptyList(void) {
      return NULL;}

boolean  empty(list l) {
      if (l==NULL) return true;
      else return false;}

element  head(list l) {
      if (empty(l)) abort();
      else return  l->value;}

list  tail(list l) {
      if (empty(l)) abort();
      else return l->next;}

list  cons(element e, list l) {
      list t;
      t = (list) malloc(sizeof(item));
      t->value=e; t->next=l; return t;}
```

```
typedef int element ;
typedef struct list_element {
      element value;
      struct list_element *next;
} node;
typedef node *list;
```

Es. 1

Si scriva una funzione ricorsiva con prototipo:

```
list unique(list l);
```

che elimini le copie di elementi duplicati adiacenti.

Ad esempio, data una lista i cui valori sono [0,2,4,6,4,2,0], applicare **`unique( l )`** produce effettivamente la stessa lista di sette elementi perché non vi sono elementi duplicati adiacenti.

Invece per la lista [0,0,2,2,4,4,6], la lista risultato dell'esecuzione di **`unique( l )`** sarà [0,2,4,6].

Es. 1 - Soluzione

Soluzione I (con uso di primitive)

```
list  unique(list l) {
   if ( empty(l)) return emptylist();
   else if ( empty( tail( l)))
          return l;
   else if ( head(l) != head( tail(l) )
          return cons( head(l), unique( tail(l) ));
   else
          return unique( tail(l) );
}
```

Es. 1 - Soluzione

Soluzione II (con accesso tramite puntatori)

```
list unique( list l ) {
   if ( l == NULL ) return NULL;
   else if ( l->next == NULL ) return l;
   else if ( l->value != l->next->value ) {
          list t;
          t= (list) malloc( sizeof(item));
          t->value = l->value;
          t->next = unique( l->next );
          return t;
   }
   else
      return unique( l->next );
}
```

Es. 3

Si scriva una funzione f() che date in ingresso due liste ordinate di interi l1 e l2, restituisca in uscita una nuova lista, ottenuta da l1 eliminando tutti gli elementi contenuti in l2.

Ad esempio, se invocata con l1=[-7,-2, 0,3,5,8,15] e l2=[-1,3,4,5], la funzione f() deve restituire la lista [-7,-2,0,8,15].

La funzione f() può essere realizzata in modo ricorsivo o iterativo, utilizzando il tipo di dato astratto list e le operazioni primitive sul tipo list definite durante il corso (che quindi possono NON essere riportate nella soluzione).

Es. 3 - Soluzione

```
list f(list l1, list l2) {

   if ( empty( l1) || empty( l2)) return l1; // fine ricorsione
   else {
         if ( head( l2) < head( l1))
              return f( l1, tail( l2));
         else if ( head( l2) == head( l1))
              return f( tail( l1), tail( l2));
         else return cons(head(l1), f(tail(l1),l2));
   }
}
```

Es. 5

Si scriva una funzione fromListToArray(), che data in ingresso una lista di interi l1, restituisca in uscita un array, creato dinamicamente, contenente gli elementi di l1.

```
int * fromListToArray(list l1, int * size)
```

Es. 5 - Soluzione

```
int * fromListToArray(list l1, int * size) {

   int * V;
   int i = 0;

   *size = length(l1);

   V = (int*) malloc(*size * sizeof(int));

   /* Primitive */
   while (! empty(l1)) {
      *(V+i) = head(l1);
      l1 = tail(l1);
      i++;
   }

   return V;
}
```

```
/* Puntatori */
while (l1!=NULL)) {
      *(V+i) = l1->value;
      l1 = l1->next;
      i++;
}
```

Es. 6

Si scriva una funzione fromArrayToList(), che data in ingresso un array di interi a, restituisca in uscita una lista, contenente gli elementi di a.

```
list fromArrayToList(int * V, int size)
```

Es. 6 - Soluzione

```
/* Primitive */
list fromArrayToList(int * V, int size) {
   if (size == 0)
     return emptylist();
   else
     return cons( *V, fromArrayToList(V+1, size-1));
}

/* Puntatori */
list fromArrayToList(int * V, int size) {
   list l;
   if (size == 0)
     return NULL;
   else{
     l=(list) malloc(sizeof(item));
     l->value=*V;
     l->next= fromArrayToList(V+1, size-1));
     return l;
   }
}
```

Es. 7

Si scriva una funzione `f()` che data in ingresso una lista ordinata di interi `l1`, restituisca in uscita una nuova lista, ottenuta da `l1` eliminando tutti gli elementi ripetuti. Ad esempio, se invocata con **l1=[1, 3, 3, 3, 5, 8, 8]**, la funzione `f()` deve restituire la lista **[1, 3, 5, 8]**.

La funzione `f()` deve essere realizzata senza l'utilizzo delle primitive.

Es. 7 - Soluzione

```
list f(list l1){
   list l;
   if (l1==NULL)
         return l1;
   else
         if ( l1->next == NULL)
              return l1;  // fine ricorsione
         else {
              if ( l1->value == l1->next->value )
                   return f( l1->next );
              else{
                   l=(list) malloc(sizeof(item));
                   l->value=l1->value;
                   l->next= f(l1->next);
                   return l;
              }
         }
}
```

Es. 8 – File binari e Stringhe

Un venditore di ortofrutta è solito fare credito ai propri clienti. Al fine di tenere traccia dei crediti, utilizza una funzione del registratore di cassa che salva, su un file binario di nome "log.dat", strutture dati del tipo `transaction` contenenti i seguenti dati:

- una stringa `customer`, contenente il nome del cliente (al più 128 caratteri, senza spazi);
- un intero `transactionId`, recante un codice identificativo della transazione;
- un float `value`, contenente l'ammontare del credito concesso.

```
#define DIM 129
typedef struct {
        char customer[DIM];
        int transactionId;
        float value;
} transaction;
```

Es. 8

Al momento di riscuotere i crediti, il commerciante deve però poter accedere ai valori registrati nel file binario.

A tal scopo, egli vuole avere un programma che, richiesto il nome del cliente, scriva su un file in formato testo gli importi relativi solo al cliente specificato.

Inoltre il file deve avere come nome il nome del cliente con l'aggiunta dell'estensione "`.txt`". Ad esempio, se il cliente richiesto si chiama "Federico", allora il file dovrà chiamarsi "`Federico.txt`". Si realizzi:

Es. 8

1. una funzione **`copy(…)`** che, ricevuti in ingresso un puntatore **`source`** al file binario, un puntatore **`dest`** al file di testo, un puntatore a carattere **`name`** e un intero **`result`** passato per riferimento, copi su **`dest`** tutti gli importi presenti in **`source`** e relativi al cliente specificato con parametro **`name`**

La funzione deve tenere traccia del numero di importi di credito copiati e deve restituire tale numero tramite il parametro **`result`**. Gli importi devono essere scritti su una sola linea, separati da uno spazio, con al termine un carattere di "newline" ('**`\n`**'). Al fine di confrontare due stringhe, si utilizzi la funzione **`strcmp(…)`**, che restituisce 0 se le due stringhe passate come parametri sono identiche

Es. 8 – Soluzione

```
void copy(FILE *source, FILE *dest, char *name, int
*result) {

      transaction temp;
      *result = 0;

      while (
        fread(&temp, sizeof(transaction), 1, source) >0
      ) {
            if (strcmp(name, temp.customer) == 0) {
                  fprintf(dest, "%f ", temp.value);
                  (*result)++;
            }
      }
      fprintf(dest, "\n");
}
```

Es. 8

2. Un programma C che chieda inizialmente all'utente il nome di un cliente. Per creare un opportuno nome per il file di destinazione, il candidato può utilizzare le funzioni di libreria:

- `strcpy(char *s, char *ct)`, che provvede a copiare la stringa ct nella stringa s;
- `strcat(char * s, char * ct)`, che concatena il contenuto della stringa ct in fondo alla stringa s (si faccia particolare attenzione a dimensionare opportunamente la stringa s per contenere i 4 caratteri dell'estensione ".txt"). La stringa s, al termine dell'invocazione, è sempre una stringa ben formata

Dopo aver aperto i file nell'opportuna modalità di lettura/scrittura, il programma utilizzi la funzione copy(…)definita al punto precedente per filtrare i dati. Il programma stampi infine a video il numero totale di crediti che sono stati copiati da un file all'altro

Es. 8 – Soluzione

```
int main() {
        char name[DIM], filename[DIM+4];
        FILE *source, *dest;
        int result = 0;

        printf("Insert customer name: ");
        scanf("%s", name);
        if ((source = fopen("log.dat", "rb")) == NULL) {
                        printf("Error opening the file %s\n", "log.dat");
                        exit(-1);}

        strcpy(filename, name);
        strcat(filename, ".txt");
        if ((dest = fopen(filename, "w")) == NULL) {
                        printf ("Error opening the file %s\n", filename);
                        exit(-1);}

        copy(source, dest, name, &result);

        fclose(source); fclose(dest);
        printf("%d records copied by log.dat to %s\n",result,filename);
        return 0;
}
```

Es. 9 – File di testo e liste

È dato un file di testo, di nome "`log.txt`", contenente una sequenza di `float` separati da spazi. Non è noto a priori quanti numeri vi siano nel file. L'obiettivo è salvare tali numeri in un `array` allocato dinamicamente, la cui dimensione corrisponda esattamente alla lunghezza della sequenza memorizzata sul file. A tale scopo:

1. si definisca una funzione `read(…)` che, ricevuto in ingresso il nome del file (puntatore a `char`), restituisca una lista contenente i valori presenti sul file (si supponga di avere a disposizione le operazioni primitive per la gestione di liste presentate a lezione)

Es. 9 - Soluzione

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

list read(char *filename) {
      list result;
      float temp;
      FILE * f;

      if ((f = fopen(filename, "r")) == NULL) {
        printf ("Error opening the file %s\n", filename);
        exit(-1); }

      result = emptylist();
      while (fscanf(f, "%f", &temp) != EOF)
            result = cons(temp, result);

      fclose(f);
      return result;
}
```

Es. 9

2. si definisca una funzione **convert(…)** che, ricevuta in ingresso una lista **l**, un puntatore **array** a una zona di memoria già allocata dinamicamente con l'opportuna dimensione, e un intero **dim** passato per riferimento, copi i valori della lista nella zona di memoria. Il numero di valori copiati deve essere restituito tramite l'intero dim

```
// versione ricorsiva…

void convert(list l, float *array, int *dim) {

      if (empty(l))
            return;
      else {
            *array = head(l);
            (*dim)++;
            convert(tail(l), ++array, dim);
      }
}
```

Es. 9

3. si definisca un programma **`main`** che legga dal file "**`log.txt`**" i valori numerici tramite la funzione **`read(…)`**; il programma deve poi allocare dinamicamente memoria sufficiente per contenere i valori letti, e tramite la funzione **`convert`**(…) deve copiare i valori della lista restituita da **`read`**(…) nell'area di memoria allocata dinamicamente. Al fine di determinare la lunghezza di una lista, il candidato può utilizzare la funzione di libreria **`int length(list)`**. Infine, il programma **`main`** deve stampare a video i valori caricati nel vettore dinamico, utilizzando la notazione sintattica tipica degli array

Il candidato ipotizzi di avere a disposizione le operazioni primitive sulle liste viste a lezione, che pertanto possono non essere riportate

Es. 9 - Soluzione

```
int main() {
      list l;
      int size, dim = 0, i;
      float *array;

      l = read("federico.txt");

      size = length(l);

      array = (float*) malloc(sizeof(float) * size);

      convert(l, array, &dim);

      for (i=0; i<size; i++)
            printf("%f\n", array[i]);

      return 0;
}
```

Es. 10

Un server Web tiene traccia degli accessi alle pagine per motivi di sicurezza. Per ogni pagina acceduta, il server scrive su un file binario una struttura dati contenente l'istante di tempo **timestamp** in cui è stata richiesta la pagina (un numero intero), l'indirizzo **ip** del client (una stringa di 15 caratteri), e la dimensione **numBytes** della pagina richiesta in byte (un numero intero)

Il numero di elementi registrati nel file non è noto a priori. L'amministratore di sistema ha la necessità di analizzare gli accessi effettuati: a tal scopo si progetti un programma che stampi a video gli istanti di accesso in base a determinati criteri

```
#define DIM 256
typedef struct {
        int timestamp;
        char ip[16];
        int numBytes;
} log;
```

Es. 10

Si realizzi:

1. una funzione `readFromFile(…)` che, ricevuti in ingresso un puntatore a un file binario di log del server e un valore intero `atLeast`, restituisca in uscita una lista contenente tutti gli istanti di accesso relativi a file la cui dimensione `numBytes` è maggiore o uguale al parametro `atLeast`. La lista restituita deve essere ordinata in senso crescente

Al fine di svolgere l'esercizio, il candidato consideri di avere a disposizione il tipo di dato astratto **lista**, e le funzioni relative (sia primitive che di alto livello), che pertanto possono non essere riportate nella soluzione

Es. 10– Soluzione punto 1

```
list readFromFile(FILE *source, int atLeast) {
   log temp;
   list result = emptylist();

   while (fread(&temp, sizeof(log), 1, source) >
   0)
      if (temp.numBytes >= atLeast)
            result = insord(temp.timestamp, result);
   return result;
}
```

Es. 10

2. un programma **`main`** che, dopo aver richiesto all'utente di inserire il nome del file e la dimensione minima, memorizzi in una lista tutti i **`timestamp`** relativi a file grandi almeno quanto indicato (utilizzando la funzione **`readFromFile`**). Dopo aver stampato a video tale lista, il programma chieda all'utente di specificare un intervallo temporale, e stampi a video i valori di **`timestamp`** contenuti nell'intervallo specificato

Es. 10– Soluzione punto 2

```
int main() {

      FILE *source;
      int start = 0, end = 0, atLeast = 0;
      char name[DIM];
      list l1, l2;

      printf("Insert file name: ");
      scanf("%s", name);
      printf("Insert min dimension: ");
      scanf("%d", &atLeast);

      if ((source = fopen(name, "rb")) == NULL) {
            printf("Error opening the file %s\n", name);
            exit(-1);
      }

      l1 = readFromFile( source, atLeast);
      fclose(source);
      …
```

Es. 10– Soluzione punto 2

```
        …
        l2 = l1;
        while (! empty(l2)) {
              printf("%d ", head(l2));
              l2 = tail(l2);
        }

        printf("Insert startTime and endTime: ");
        scanf("%d%d", &start, &end);

        while((! empty(l1)) && (head(l1) <= end)) {
              if (head(l1) >= start)
                    printf("%d ", head(l1));
              l1 = tail(l1);
        }

        return 0;
}
```

Stack: vettore vs. puntatori

Definizioni e Primitive

Vettore

```
#include "element.h"
#define MAX 1024
typedef struct {
        element val[MAX];
        int sp;
} st;
typedef st * stack;
```

Puntatori

```
#include "element.h"
typedef struct stackN {
        element value;
        struct stackN *next;
} stackNode;
typedef stackNode *stack;
```

Primitive caso Vettore

```
void push(element, stack);
element pop(stack);
```

Primitive caso Puntatori

```
void push(element, stack*);
element pop(stack*);
```

Primitive comuni

```
stack newStack(void);
boolean
isEmptyStack(stack);
boolean isFullStack(stack);
```

Es. 14 – Stack
funzione reverse()

Realizzare una funzione

```
... reverse (...);
```

che, dato come argomento in ingresso uno stack, restituisca un altro stack con gli elementi in ordine inverso.

Si realizzino due versioni di tale funzione: la prima utilizzando la definizione a vettore senza primitive, la seconda utilizzando le primitive della definizione a puntatori.

Es. 14 – Stack
funzione reverse()

Vettore

```
stack reverse (stack s) {
   stack newS = newstack();
   int i;
   for(i=0; i < s->sp; i++){
    newS->val[i] = s->val[sp-i-1];
   }
   newS->sp=i;
   return newS;
}
```

Es. 14 – Stack
funzione reverse()

Primitive caso Puntatori (funzione iterativa)

```
void reverse (stack * s) {
   /* creo un nuovo stack vuoto */
   stack newS = newStack();

   /* itero finché non svuoto lo stack originario */
   while( !emptyStack(*s) )
         /* inserisco in fondo allo stack nuovo gli elementi
            che erano in cima allo stack originario */
         push( pop(s), &newS);

   /* restituisco il nuovo stack per riferimento */
   *s=newS;
return;
}
```

Versione distruttiva: alla fine della reverse() lo stack originario non è più disponibile

Es. 15 – Stack
funzione dim()

Realizzare una funzione

```
int dim (stack s);
```

che conti il numero di elementi presenti sullo stack.
Si realizzino tre versioni di tale funzione: una utilizzando la definizione a vettore, una utilizzando la definizione a puntatori ed una utilizzando le primitive del caso a puntatori.

Es. 15 – Stack
funzione dim()

Vettore

```
int dim (stack s) {
   return s->sp;}
```

Puntatori (ricorsivo)

```
int dim (stack s) {
   if (s == NULL) return 0;
   else return 1 + dim(s->next);}
```

Puntatori (iterativo)

```
int dim (stack s) {
   int i;
   stack temp = s;
   for(i=0; temp != NULL; i++) temp=temp->next;
   return i;}
```

Es. 15 – Stack
funzione dim()

Primitive caso Puntatori

```
int dim (stack s) {
   element el;
   int val;
   if (emptyStack(s)) return 0;
   else{
     el=pop(&s);
     val=1+dim(s);
     push(el,&s);
     return val;
   }
}
```

Coda FIFO
definizione e primitive

```
typedef struct queue_element {
        element value;
        struct queue_element * next;
} queueNode;
typedef  queueNode * startQueue;
typedef  queueNode * endQueue;
```

```
int isEmptyQueue (startQueue aQueue);
void createEmptyQueue (startQueue * start, endQueue * end);
void enQueue (element el, startQueue * start, endQueue * end);
element deQueue (startQueue * start, endQueue * end);
```

Es. 16 – Code FIFO
funzione elementAt()

Realizzare una funzione

```
element elementAt (int n, startQueue * start);
```

che restituisca l'n-esimo elemento della coda.

Si realizzino due versioni di tale funzione: una utilizzando la rappresentazione a puntatori, l'altra utilizzando le primitive.

Es. 16 – Code FIFO
funzione elementAt()

Rappresentazione a puntatori

```
element elementAt (int n, startQueue * start){
   startQueue* temp=start;
   int i;
   /* avanzo di n valori */
   for(i=1; (i<n) && ! (*temp==NULL); i++){

                        /* avanzo al queueNode successivo */

   *temp =(*temp)->next;
   }


    /* se ci sono meno di n valori restituisco 0 */
       if(*temp==NULL) return 0; /* coda vuota */
/* altrimenti restituisco l'n-esimo elemento */
     return (*temp)->value;
}
```

Es. 16 – Code FIFO
funzione elementAt()

Primitive

```
element elementAt (int n, startQueue* start, endQueue* end){
   startQueue newStart; endQueue newEnd;
   element el, ret=0; int i;
   createEmptyQueue(&newStart, &newEnd);

   if(isEmptyQueue(start)) return 0; /* coda vuota */

   /* itero lungo tutta la coda */
   for(i=1; !isEmptyQueue(start); i++){
   /* le primitive sono distruttive, quindi gli elementi
    tolti da una coda devono essere inseriti in un'altra */
      el=deQueue(start,end);
      if(i==n) ret=el;
      enQueue(el,&newStart,&newEnd);
   }

   /* Assegno alla coda originaria i valori della nuova coda */
   *start=newStart; *end=newEnd;
   return ret; }
```