Sequenze

Sommario

- Le strutture dati elementari per implementare sequenze:
  - Vettori
  - Liste

Strutture dati elementari

- Le strutture dati vettore e lista sono fra le strutture dati più usate e semplici
- Il loro scopo è quello di permettere l'accesso ai membri di una collezione generalmente omogenea di dati
- Per alcuni linguaggi di programmazione sono addirittura primitive del linguaggio (vettori in C/C++ e liste in LISP)
- Sebbene sia possibile realizzare l'una tramite l'altra, i costi associati alle operazioni di inserzione e cancellazione variano notevolmente nelle diverse implementazioni

Vettori

- Un vettore è una struttura dati che permette l'inserimento di dati e l'accesso a questi tramite un indice intero
- In C/C++ l'indice parte obbligatoriamente da 0
- Nella maggior parte degli elaboratori vi è una corrispondenza diretta con la memoria centrale (questo implica alta efficienza)

- **Vantaggi:** Il tempo impiegato per un inserimento o un accesso non dipende dall'indice
- **Svantaggi:** Per ragioni di efficienza la memorizzazione avviene in aree contigue di memoria, ma questo impedisce la cancellazione o l'inserimento di elementi in posizioni intermedie

Esempio: Crivello di Eratostene

```
int main(){
    const int N = 1000;
    int i, a[N];

    //inizializzazione a 1 del vettore
    for (i = 2; i < N; i++)
       a[i] = 1;

     for (i = 2; i < N; i++)
       if (a[i]) //se numero primo elimina tutti multipli
         for (int j = i; j*i < N; j++) a[i*j] = 0;

     //stampa
    for (i = 2; i < N; i++)
       if (a[i]) cout << " " << i;
     cout << endl;

    return 0;
}
```

Idea intuitiva del Crivello di Eratostene

- Si prende un vettore di N elementi a 1
- Si parte dal secondo elemento e si cancellano (mettono a 0) tutti gli elementi di posizione multipla di 2
- Si considera l'elemento successivo che non sia stato cancellato
- Questo elemento non è divisibile per alcun numero precedente (altrimenti sarebbe stato messo a 0) e deve pertanto essere primo
- Si cancellano tutti i suoi multipli

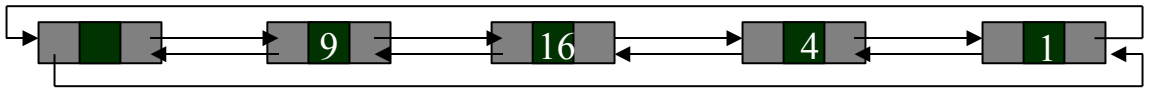
Liste

- Una lista concatenata è un insieme di oggetti e...
- ..ogni oggetto è inserito in un nodo che contiene anche un link (un riferimento) ad un altro nodo

- **Uso:** quando è necessario scandire un insieme di oggetti in modo sequenziale
- **Vantaggi:** frequenti operazioni di cancellazione o inserzioni
- **Svantaggi:** si può accedere ad un elemento di posizione *i* solo dopo aver acceduto a tutti gli *i-1* elementi precedenti

Liste

- Di norma si pensa ad una lista come ad una struttura che implementa una disposizione sequenziale di oggetti
- In linea di principio tuttavia l'ultimo nodo potrebbe essere collegato con il primo: in questo caso avremo una lista *circolare*



Liste

- Una lista può essere:
  - concatenata semplice: un solo link
  - concatenata doppia (bidirezionale): due link

- Le liste bidirezionali hanno un link al nodo che le precede nella sequenza ed uno al nodo che le segue
- Con le liste concatenate semplici non è possibile risalire al nodo precedente ma si deve nuovamente scorrere tutta la sequenza
- Le liste concatenate doppie occupano più spazio in memoria delle liste concatenate semplici

Convenzioni

- In una lista si ha sempre un nodo detto testa ed un *modo convenzionale* per indicare la fine della lista
- La testa di una lista semplice non ha predecessori
- I tre modi convenzionali di trattare il link del nodo dell'ultimo elemento sono:
  - link nullo
  - link a nodo fittizio o sentinella
  - link al primo nodo (lista circolare)

Implementazione C++

- L'implementazione in C/C++ di una lista fa uso dei puntatori:

```
struct Node {
    int key;
    Node * next;
};

struct Node {
    int key;
    Node * next;
    Node * prec;
};
```

Esempio: Problema di Giuseppe Flavio

```
struct Node{ int item; Node* next;};
void main(int argc, char * argv[]){
    int i, N = atoi(argv[1]), M = atoi(argv[2]);
    Node * t = new Node;
    t->item=1;
    t->next = t;
    Node * x = t;
    for (i = 2; i <= N; i++){ //creazione della lista
        x->next = new Node;
        x->next->item=i;
        x->next->next=t;
        x=x->next;
        }
    while (x != x->next){ //eliminazione
        for (i = 1; i < M; i++)
            x = x->next; //spostamento
        x->next = x->next->next;
     }
     cout << x->item << endl;//stampa l'ultimo elemento
}
```

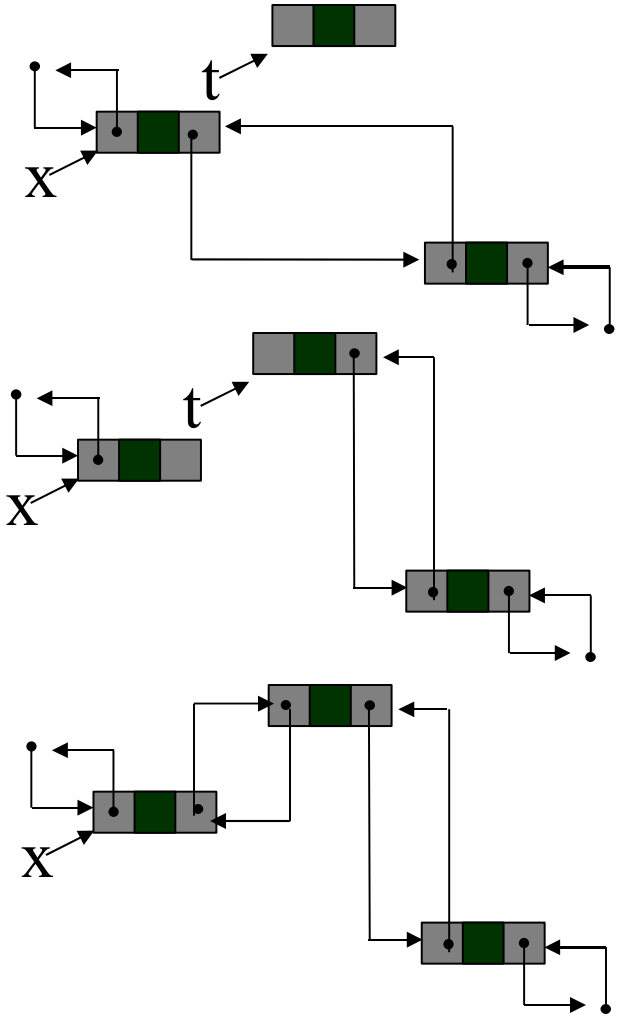
Idea intuitiva del Problema di G. Flavio

- Si parte da una lista circolare di N elementi
- Si elimina l'elemento di posizione M dopo la testa
- Ci si muove a partire dall'elemento successivo di M posizioni e si elimina il nodo corrispondente
- Vogliamo trovare l'ultimo nodo che rimane

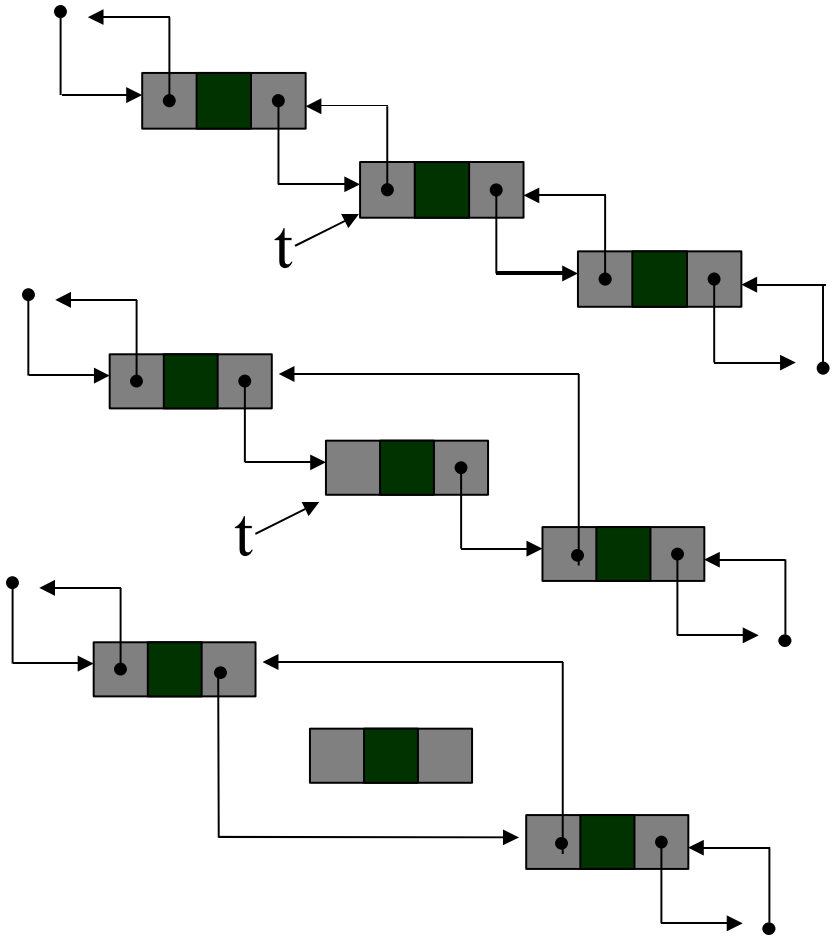
Operazioni definite sulla lista

- Per una lista si possono definire le operazioni di:
  - inserimento
  - cancellazione
  - ricerca
- Di seguito se ne danno le rappresentazioni grafiche e le implementazioni in pseudocodice per una lista bidirezionale

Rappresentazione grafica inserzione



Rappresentazione grafica cancellazione



Inserimento in testa

```
List-Insert(L,x)
1  next[x]←head[L]
2  if head[L] ≠ NIL
3       then prev[head[L]]←x
4  head[L]←x
5  prev[x]←NIL
```

- **Nota:** il test alla riga 2 serve per prendere in considerazione il caso di inizializzazione (inserimento in lista vuota)

Cancellazione

```
List-Delete(L,x)
1  if prev[x] ≠ NIL
2       then next[prev[x]]←next[x]
3       else head[L]←next[x]
4  if next[x] ≠ NIL
5       then prev[next[x]]←prev[x]
```

Memory leakage

- Quando si cancella un nodo si deve porre attenzione alla sua effettiva deallocazione dallo heap
- Nel caso in cui si *elimini* un nodo solamente rendendolo inaccessibile non si libera effettivamente la memoria (memory leakage)
- Se vi sono molte eliminazioni si può rischiare di esaurire la memoria disponibile per il programma

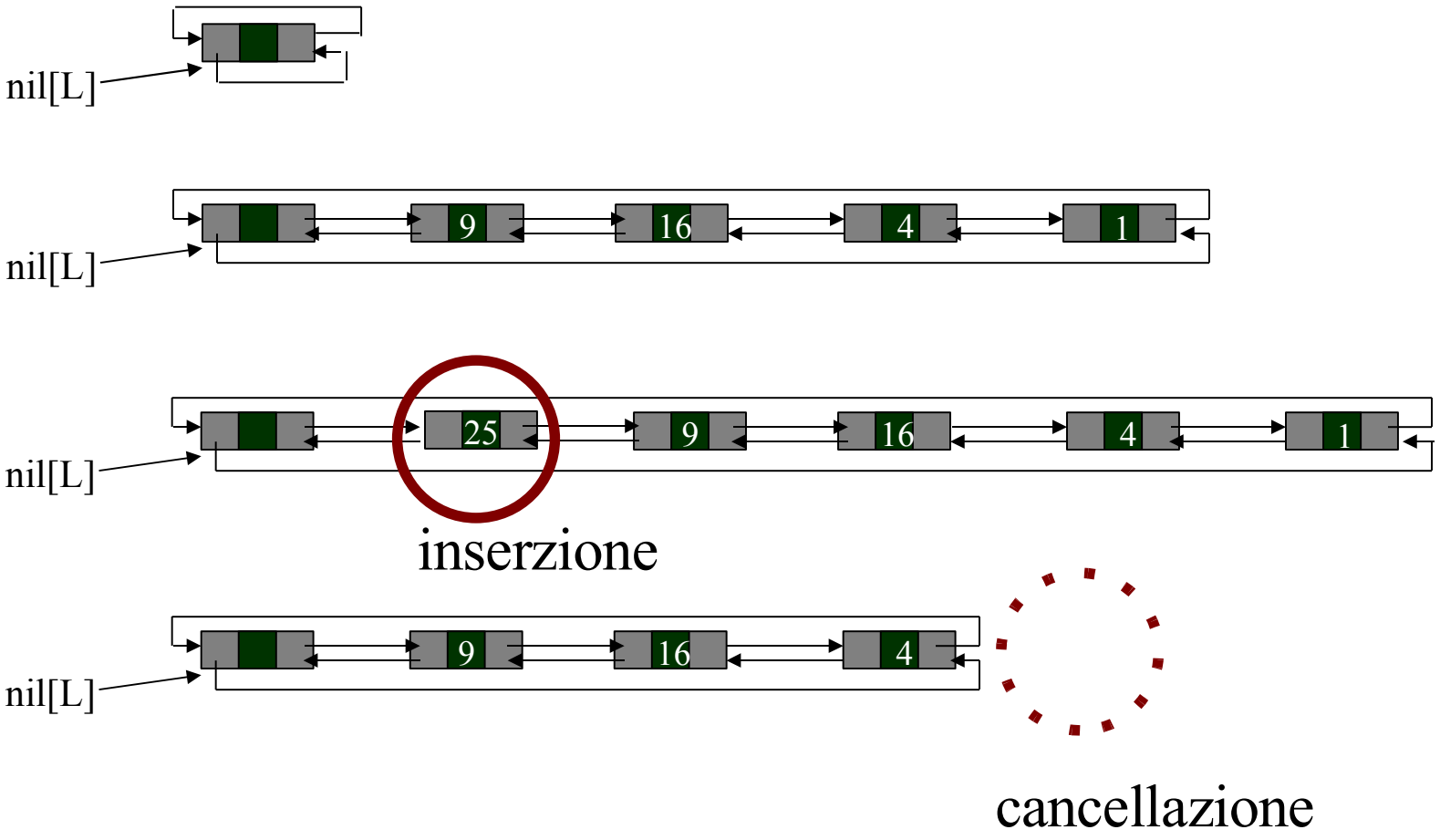
Ricerca

```
List-Search(L,k)
1  x←head[L]
2  while x ≠ NIL and key[x] ≠ k
3       do x ← next[x]
4  return x
```

La sentinella

- Si può semplificare la gestione delle varie operazioni se si eliminano i casi limite relativi alla testa e alla coda
- Per fare questo si utilizza un elemento di appoggio detto NIL[L] che sostituisca tutti i riferimenti a NIL
- Tale elemento non ha informazioni significative nel campo key ed ha inizialmente i link next e prev che puntano a se stesso

Rappresentazione Grafica



Implementazioni con sentinella

```
List-Insert(L,x)
1  next[x]←next[nil[L]]
2  prev[next[nil[L]]]←x
3  next[nil[L]]←x
4  prev[x]←nil[L]
```

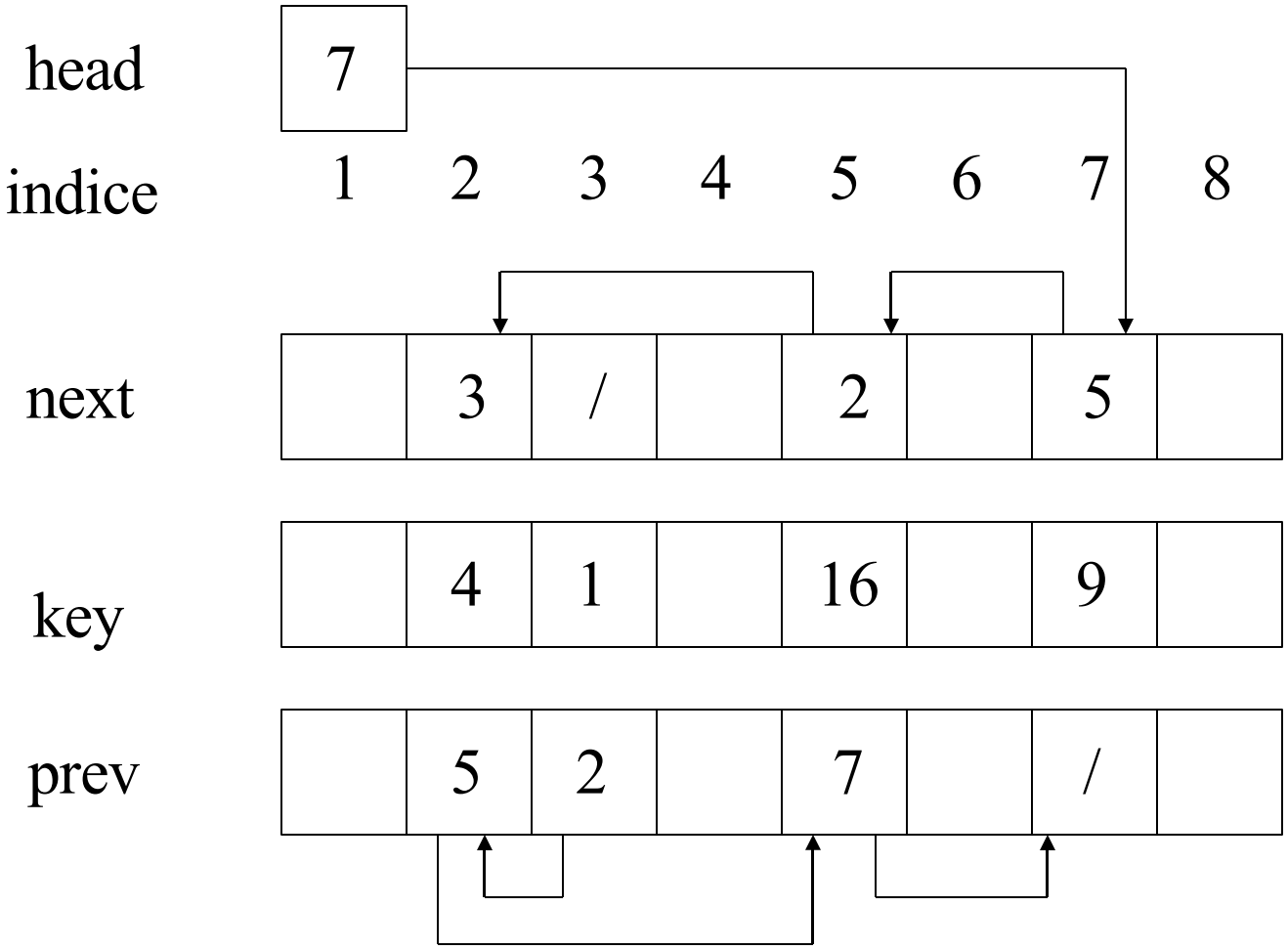
```
List-Delete(L,x)
1  next[prev[x]]←next[x]
2  prev[next[x]]←prev[x]
```

```
List-Search(L,k)
1  x←next[nil[L]]
2  while x ≠ nil[L] e key[x] ≠ k
3       do x ← next[x]
4  return x
```

Implementazione di lista con più vettori

- Si può rappresentare un insieme dei oggetti che abbiano gli stessi campi con un vettore per ogni campo
- Per realizzare una lista concatenata si possono pertanto utilizzare tre vettori: due per i link e uno per la chiave
- Un link adesso è solo l'indice della posizione del nodo puntato nell'insieme di vettori
- Per indicare un link nullo di solito si usa un intero come 0 o -1 che sicuramente non rappresenti un indice valido del vettore

Esempio



Nota

- L’uso nello pseudocodice della notazione:

  next[x]

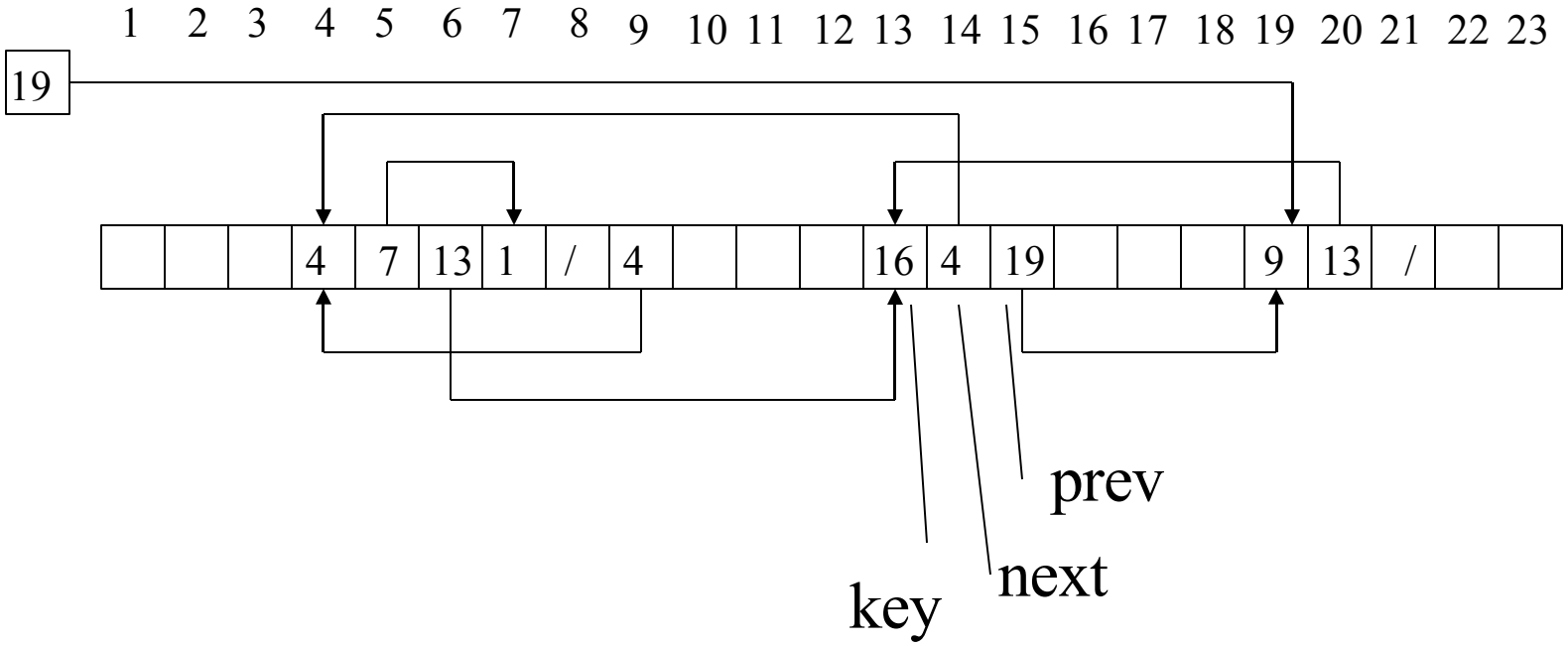
  prev[x]

  key[x]
- ...corrisponde proprio alla notazione utilizzata nella maggior parte dei linguaggi di programmazione per indicare l’implementazione vista

Implementazione lista con singolo vettore

- La memoria di un calcolatore può essere vista come un unico grande array.
- Un oggetto è generlamente memorizzato in un insieme contiguo di celle di memoria, ovvero i diversi campi dell'oggetto si trovano a diversi scostamenti dall'inizio dell'oggetto stesso
- Si può sfruttare questo meccanismo per implementare liste in ambienti che non supportano i puntatori:
  - il primo elemento contiene la key
  - il secondo elemento l'indice del next
  - il terzo elemento l'indice del prev

Esempio


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
19
4 7 13 1 / 4 16 4 19 9 13 /
prev
next
key