

# Gestione della memoria

Stack di attivazione, Heap

# La gestione della memoria

Come il compilatore-interprete organizza i dati necessari all'esecuzione del programma.

Alcuni aspetti organizzativi già visti nel corso in assembly

Il codice macchina, sia

- scritto in assembly, sia
- generato da compilatori,

struttura la memoria in modo simile

# Uso della Memoria RAM

Nell'uso tipico, il codice ARM prevede la divisione della memoria nei seguenti intervalli consecutivi.

- 0 - 0xFFF: riservata al sistema operativo
- 0x1000 - ww : codice programma (.text)
- ww - xx : costanti, variabili **programma principale** (.data)
- xx - yy : **heap** per dati dinamici (liste, alberi)
- yy - zz : **stack** per chiamate di procedura, stack di attivazione

I dati del programma distribuiti in zone della memoria

Il registro r13, sp, (**stack pointer**) punta alla cima dello stack

Il registro r11, fp, (**frame pointer**) punta al "frame" della procedura in esecuzione

# Tipi di allocazione della memoria

Tre meccanismi di allocazione della memoria:

- **statica**: memoria allocata a tempo di compilazione
- **dinamica**: memoria allocata a tempo d'esecuzione, divisa in:
  - **pila (stack)**: oggetti allocati con politica LIFO
  - **heap**: oggetti allocati e de-allocati in qualsiasi momento

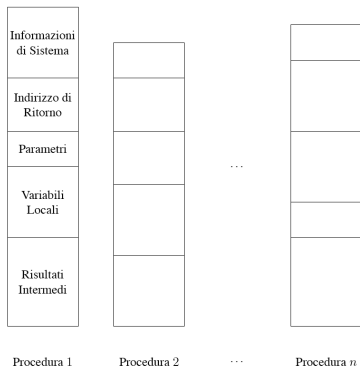
Normalmente un programma usa tutti e tre i meccanismi

# Allocazione statica

- Gli oggetti hanno un **indirizzo assoluto**, mantenuto per tutta la durata dell'esecuzione.
- Solitamente sono allocati staticamente:
  - variabili globali
  - alcune variabili locali alle procedure dichiarate **static** in C, C++, PHP  
un'unica istanza per tutte le chiamate  
es. counter, pseudo-random
  - campi condivisi tra tutti gli oggetti di una classe, **static** in Java, C#, Python, Ruby
  - costanti determinabili a tempo di compilazione
  - tabelle usate dal supporto a run-time  
(per type checking, garbage collection, ecc.)
- Alcuni linguaggi di programmazione (vecchie versioni Fortran), usano l'allocazione statica per **tutti i dati**, in questo caso:

# Allocazione statica del programma

- ad ogni variabile, locale o globale, vi assegnato un indirizzo fisso



- le variabili locali di una procedura mantengono il valore anche dopo la fine della procedura

# Svantaggi dell'allocazione statica completa.

Non permette la ricorsione:

- Per ciascuna chiamata ricorsiva di una stessa procedura devo memorizzare:
  - variabili locali, ogni chiamata ricorsiva fa riferimento ad una sua istanza,
  - informazioni di controllo (indirizzo di ritorno).

Si usa più spazio del necessario:

- si allocare spazio per tutte le variabili, di tutto il codice
- di volta in volta, solo una piccola parte di queste sono attive (quelle associate alle procedure aperte)
- problema alleviato dalla memoria virtuale

Non permette strutture dati dinamiche: es. liste, alberi.

**Vantaggi:** accesso diretto, veloce, a tutte le variabili.

# Stack di attivazione

Parte di memoria gestita come una **pila** destinata contenere i **dati locali alle procedure**

- ad ogni chiamata, allocato dello spazio, **record di attivazione**
- spazio de-allocato all'uscita dalla procedura

Si sfrutta il fatto che viene chiusa sempre l'ultima funzione chiamata

Gestione efficiente della memoria, non si creano frammentazione



# Record di attivazione

Ogni istanza di procedura in esecuzione possiede un **record di attivazione** (RdA o frame), spazio di memoria per contenere:

Dati:

- variabili locali
- parametri in ingresso e in uscita
- risultati intermedi

Informazione di controllo

- indirizzo di ritorno
- link dinamico (al frame della procedura chiamante)
- link statico (al frame della procedura genitore) (non sempre presente)
- salvataggio dei registri

# Formato di un record di attivazione

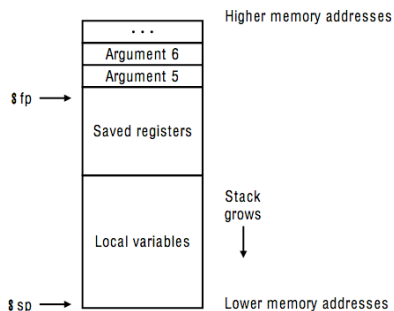


Immagine relativa all'assembly ARM

# Record di attivazione associati a blocchi di comandi

Ogni blocco, con dichiarazione, **può avere** un suo record di attivazione (più semplice, meno informazioni di controllo)

- variabili locali
- risultati intermedi
- link dinamico (al frame della procedura chiamante)

# Stack di attivazione:

Pila (LIFO) contenente i RdA

struttura dati naturale per gestire i RdA

- l'ultima procedura chiamata è la prima a terminare

## Esempio: gestione della memoria con blocchi anonimi

```
{ int x=0;
  int y=x+1;
  { int z=(x+y)*(x-y);
  };
};
```

- creo un record con spazio per  $x$ ,  $y$
- assegna valori di  $x$ ,  $y$ 
  - creo un record per blocco interno spazio per  $z$  e, eventualmente, risultati intermedi
  - assegna valore per  $z$
  - elimino record per il blocco interno
- elimino record per il blocco esterno

Nota: nel blocco interno l'accesso alle variabili non locali  $x$  e  $y$ , vanno cercate in blocchi esterni (risalire la catena dinamica)

# Accesso ai dati nel RdA attuale

L'indirizzo dei dati (variabili, risultati intermedi) **locali** al blocco in esecuzione deve **essere calcolato**

Il compilatore può determinare solo l'**offset**,  
posizione relativa del dato rispetto all'indirizzo base del record\_\_ a  
tempo di compilazione non nota la posizione del record di attivazione

Nel calcolo si usa il **Frame Pointer (FP)**:

- variabile (registro) contiene l'indirizzo base dell'ultimo frame (RdA)
- i dati dell'ultimo frame sono accessibili per **offset** rispetto allo FP:
  - $\text{indirizzo\_dato} = \text{FP} + \text{offset}$
  - offset determinato staticamente, a tempo di compilazione

Per dati **non locali** al blocco in esecuzione

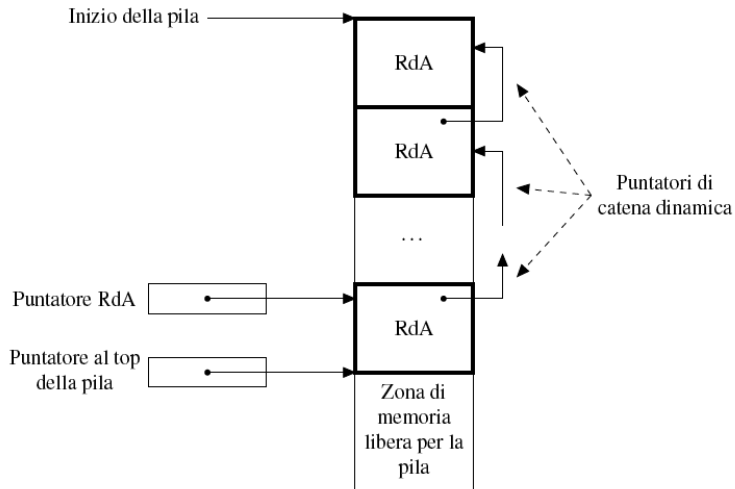
- necessario determinare l'indirizzo base del RdA del dato, uso
  - **Link statico**, puntatore alla Catena Statica
    - puntatore a un precedente record sullo stack
  - **Link dinamico**, puntatore alla Catena Dinamica
    - puntatore al precedente record sullo stack
- si risale la catena dei link
- si determina l'indirizzo base del RdA contenente la variabile
- $\text{indirizzoDato} = \text{indirizzoBase} + \text{offset}$

Per la gestione dello stack di attivazione si usa anche

- Stack Pointer (SP)
  - punta alla fine della pila, primo spazio di memoria disponibile



# Gestione dello stack di attivazione



Operazioni per la gestione (aggiornamento di FP, SP e link dinamici)

- Ingresso nel blocco: Push  
allocazione dello spazio e scrittura dei link
  - link dinamico del nuovo RdA := FP
  - $FP = SP$
  - $SP = SP + \text{dimensione nuovo RdA}$
- Uscita dal blocco: Pop, Elimina l'ultimo RdA  
recupera spazio e ripristina puntatori
  - $SP = FP$
  - $FP := \text{link dinamico nel RdA rimosso dallo stack}$

In molti linguaggi si preferisce evitare l'implementazione a pila per i blocchi anonimi:

- tutte le dichiarazioni dei blocchi annidati sono raccolte dal compilatore
- viene allocato spazio per tutti i blocchi
- spreco di memoria
- maggiore velocità: meno azioni sulla pila

## Chiamata procedura:

- allocazione RdA
- passaggio parametri
- inizializzazione variabili locali
- gestione informazioni di controllo: link statici, dinamici
- registri: salvataggio

## Uscita procedura:

- passaggio risultato
- gestione informazioni di controllo:
- registri: ripristino
- de-allocazione RdA

# Gestione delle chiamate di procedura

La gestione della pila è compiuta mediante:

- sequenza di chiamata - chiamante
  - prologo - chiamato
  - 
  - epilogo - chiamato
- sequenza di ritorno - chiamante

La ripartizione tra chiamante e chiamato è in parte arbitraria

- inserire il codice nella procedura chiamata porta a generare meno codice
  - istruzioni presenti **una volta sola** al posto di **tante volte quanti** i punti di chiamata alla procedura

# Esempio uso stack di attivazione, ricorsione

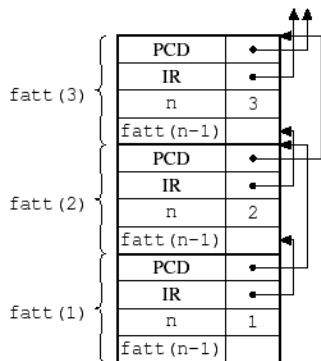
```
int fact (int n) {  
    if (n<= 1) return 1;  
    else return n * fact(n-1);  
}
```

Nel RdA troviamo:

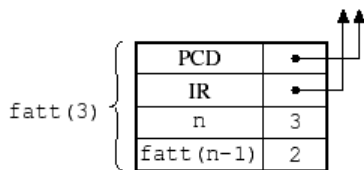
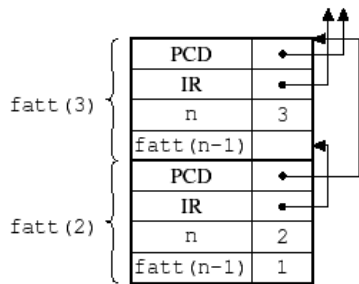
- parametri di ingresso: n
- parametri in uscita - risultato: fact(n)
- link statici, dinamici e indirizzo di ritorno, backup registri processore

Tanti RdA quanti il valore iniziale di n

# Fattoriale evoluzione dello stack



# Fattoriale evoluzione dello stack





L'accesso ai dati non locali nello stack di attivazione dipende dalle politiche di scope

- Scope statico:
  - catena statica
  - display
- Scope dinamico:
  - solo RdA
  - A-list
  - Tabella centrale dell'ambiente (CRT)

# Scope statico: presentazione del problema

Consideriamo il programma:

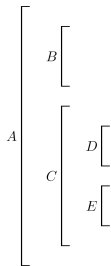
```
int x=10;
void incx () {
    x=x+1;    }
void foo (){
    int x=0;
    incx();  }
foo();
incx();
```

- `incx` viene chiamato due volte: indirettamente tramite `foo`, direttamente;
- `incx` accede sempre alla stessa variabile `x`
- `x` è memorizzato in un certo RdA
- problema: determinare di quanti RdA scendere nella pila

# Scope statico: legami validi

I legami validi sono quelli definiti:

- nella procedura corrente
- nella procedura genitore
- nel genitore del genitore . . . .



- In D, i legami validi sono quelli definiti in D, C, A
- le altre dichiarazioni non sono visibili

A **tempo di compilazione**, per ogni variabile  $x$  usata in una procedura  $P$  si determina:

- quale procedura **antenato**  $Q$  contiene la dichiarazione di  $x$  a cui far riferimento,
- di quanti **livelli**  $Q$  è antenato di  $P$
- l'offset di  $x$  nei record di attivazione di  $Q$

A **tempo di esecuzione**, nella procedura  $P$  per accedere a  $x$  si deve

- accedere all'ultimo RdA di  $Q$  attivo
- dall'indirizzo base del RdA di  $Q$ , si determina posizione di  $x$

# Link statico

Per poter accedere ai RdA degli antenati:

- ogni RdA contiene un puntatore all'ultimo RdA del genitore.
- chiamato **link statico**:

I link statici determinano **catena statica**: la lista dei RdA degli antenati

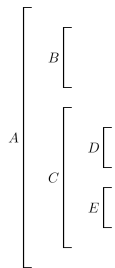
Nota: per ogni variabile  $x$  non locale ad una procedura  $D$ :

- il numero di link statici da seguire per trovare il RdA contenente  $x$  è determinabile a tempo di compilazione, resta costante in ogni ricerca
- la posizione relativa di  $x$  (offset) nel RdA è fissa:
  - a tempo di compilazione si determina l'offset
  - non serve memorizzare il nome delle variabili negli RdA

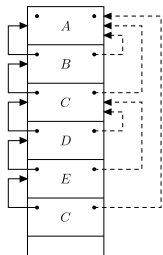
# Riassumendo:

- link dinamico (**procedura chiamante**) dipende dalla sequenza di esecuzione del programma,  
definisce la **catena dinamica**,
- link statico (**procedura genitore**) dipende dalla struttura del codice  
definisce la **catena statica**

# Esempio: programma con struttura a blocchi



La sequenza di chiamate A, B, C, D, E, C, genera la pila:



# Esempio esecuzione.

```
int x = 10;
void A(){
    x=x+1;}
void B(){
    int x = 5;
    void C (){
        x=x+2; A();
    }
    A(); C();
}
B();
```



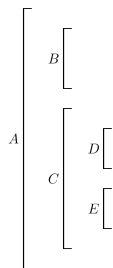
# Come si crea il link statico?

La procedura chiamante  $C_h$  determina il link statico della procedura chiamata  $S$ ,

- lo passa ad  $S$ , o lo inserisce nel RdA di  $S$
- come  $C_h$  determina il link statico da passare a  $S$ ?  
coincide con l'indirizzo base del RdA della procedura dove  $S$  è definita
  - si ripetono i calcoli necessari per accedere a una variabile dichiarata nel blocco di  $S$
  - quando  $C_h$  chiama  $S$  sa se la definizione di  $S$  è:
    - nelle dichiarazioni di  $C_h$  (profondità  $k=0$ )  
passa il **Frame pointer** come link statico
    - nelle dichiarazioni di un  $n$ -esimo avo di  $C_h$ , (profondità  $k = n$ ) percorre la catena statica per  $n$  passi, per determinare il link statico da passare
    - in altre posizioni  $S$  non sarebbe visibile

# Esempi di profondità relativa k

Nel caso di un programma con struttura



la procedura D può chiamare: A, B, C, D, E, ciascuna con profondità relativa K:

(A,2); (B,1); (C,1); (D,0); (E,0)

k determina quanti passi percorrere nella catena statica per trovare

- il link statico da passare a una procedura chiamata
- l'indirizzo base per determinare la locazione di una variabile,

# Come si determina il link statico?

- Se  $k=-1$ ,  $S$  definita in  $Ch$ :
  - $Ch$  passa a  $S$  il suo frame pointer
  -
- Se  $k=0$ :
  - $Ch$  passa a  $S$ , come link statico il suo link statico
- Se  $k>0$ :
  - $Ch$  scende la catena statica di  $k$  passi, passa a  $S$  il link statico trovato

# Costi di accesso ai dati non locali.

L'accesso a variabili non locali comporta la scansione della catena di link statici,

- costo proporzionale alla profondità.
- si può ridurre questo costo a un singolo accesso usando il **display**:

Un unico array contenente il link ai RdA visibili al momento attuale

- l' $i$ -esimo elemento dell'array: puntatore al RdA del sottoprogramma
  - di livello di annidamento statico (profondità assoluta)  $i$ :
    - programma principale: ann. statico 0
    - procedure definite nel programma principale: ann. stat. 1
    - ...
  - ultima istanza attivata (se, per ricorsione, ci sono più istanze)

Per accedere a una variabile  $x$  con annidamento statico  $i$

- l' $i$ -esimo elemento nel display determina il RdA da usare

# Come si aggiorna il display

Ad ogni ingresso e uscita di procedura.

La procedura chiamata  $P$ , nel preambolo:

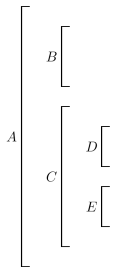
- esamina nel display la locazione relativa al suo annidamento statico
  - salva (nel suo RdA) il valore contenuto
  - e lo sostituisce con il puntatore al suo RdA

Al termine della chiamata,  $P$  ripristina il valore originario

Algoritmo semplice, correttezza non banale:

- ogni procedura, all'ingresso aggiorna il display in un'unica posizione  
il display modificato è corretto per l'ambiente della procedura
- alla sua uscita ripristina il valore originario  
il display torna ad essere corretto per il chiamante

# Esempio



Es. Sequenze di chiamate A C D C E

Statisticamente, nel codice, raramente si percorrono più di 3 passi nella catena statica

Display poco usato nelle implementazioni moderne



Con lo scope dinamico l'associazione nome-oggetto denotabile dipende

- dal flusso del controllo a run-time, ossia dall'ordine con cui i sottoprogrammi sono chiamati

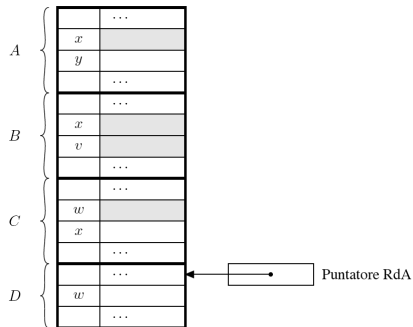
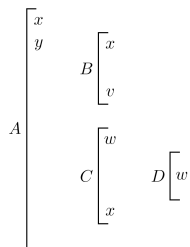
La regola generale è semplice:

- il legame valido per il nome **n**
- è determinato dall'ultima dichiarazione del nome **n** eseguita

# Implementazione ovvia

- **memorizzare** i nomi, e i relativi tipi, negli RdA
  - a differenza dello scope statico dove non è necessario memorizzarli
- ricerca del nome scandendo, a scendere, lo stack di attivazione

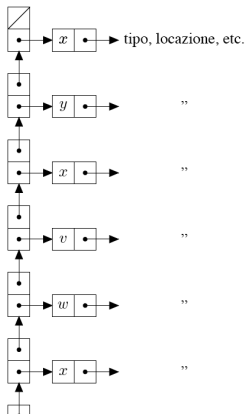
# Esempio - Programma con chiamate A,B,C,D



# Variante: Association List: A-list

Le associazioni (nome, valore) sono raggruppate in una struttura apposita

- una lista di legami validi
- aggiornata insieme allo Stack di Attivazione



# Costi delle A-list

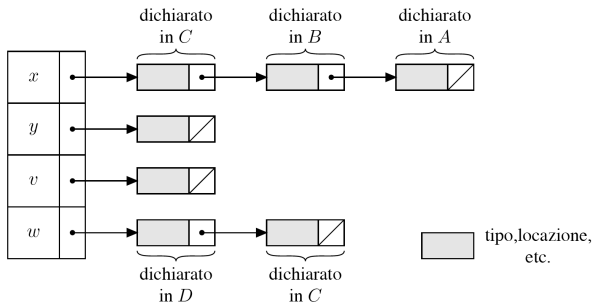
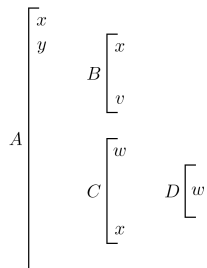
- Molto semplice da implementare
- Occupazione di memoria:
  - nomi presenti esplicitamente
- Costo di gestione
  - ingresso/uscita da blocco
    - inserzione/rimozione di blocchi sulla pila
- Tempo di accesso
  - **lineare** nella profondità della A-list

Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco. . .

# Tabella centrale dei riferimenti, CRT

- Evita le lunghe scansioni della A-list
- Una tabella mantiene tutti i nomi distinti del programma
  - se nomi noti staticamente, accesso in tempo costante
  - altrimenti, accesso hash
- Ad ogni nome è associata la lista delle associazioni di quel nome:
  - la più recente è la prima
  - le altre, disattivate e per uso futuro, seguono
- accesso agli identificatori in tempo costante

# Esempio (con chiamate A,B,C,D )

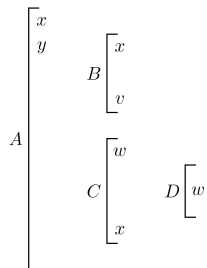


## Seconda possibile implementazione

- Tabella attuale,
- Pila dei legami sospesi,  
una singola pila in cui inserisco i legami nascosti,  
da aggiornare in ingresso e uscita dai blocchi.



# Esempio (con chiamate A,B,C,D )



A

x	1	$\alpha_1$
y	1	$\alpha_2$
v	0	-
w	0	-

AB

x	1	$\beta_1$
y	1	$\alpha_2$
v	1	$\beta_2$
w	0	-

ABC

x	1	$\gamma_1$
y	1	$\alpha_2$
v	0	$\beta_2$
w	1	$\gamma_2$

ABCD

x	1	$\gamma_1$
y	1	$\alpha_2$
v	0	$\beta_2$
w	1	$\delta_1$

x	$\alpha_1$
---	------------

x	$\alpha_1$
x	$\beta_1$

x	$\alpha_1$
x	$\beta_1$
w	$\delta_1$

# Costi della CRT

- Gestione più complessa rispetto all'A-list

## Costo di gestione

- ingresso/uscita da blocco
  - manipolazione delle liste dei nomi dichiarati nel blocco

## Tempo di accesso

- costante (due accessi indiretti)

## Confronto con A-List

- si riduce il tempo d'accesso medio,
- si aumenta il tempo di ingresso/uscita dal blocco

# Allocazione dinamica dati nel heap

- **Heap**: zona di memoria le cui parti (blocchi) possono essere allocate (e de-allocate) a seconda della necessità
- Necessario quando il linguaggio permette:
  - creazioni di oggetti attraverso istruzioni, come chiamate a 'new'
  - tipi di dato dinamici: liste, alberi
  - oggetti di dimensioni variabili: vettore a dimensione variabile
  - oggetti che sopravvivono alla procedura che li ha creati
- Gestione dello heap:
  - allocare spazio libero
  - recuperare spazi di memoria non utilizzati
- Problemi:
  - gestione efficiente dello spazio: frammentazione
  - velocità di ricerca spazi liberi

# Implementazione: Heap in blocchi di dimensione fissa

- Inizialmente: tutti i blocchi collegati nella lista libera
- allocazione di uno o più blocchi
- de-allocazione: restituzione alla lista libera
- il vincolo della dimensione fissa rende il meccanismo troppo rigido:
  - non fornisce blocchi di elevata dimensione per strutture dati contigue di dimensione elevata
  - non posso implementare `malloc` di C

# Implementazione: Heap in blocchi di dimensione variabile

- Inizialmente: lista libera costituita da un unico blocco, poi lista formata da blocchi di dimensione variabile
- allocazione: determinare un blocco libero della dimensione opportuna
  - che viene diviso in:
    - parte assegnata
    - resto blocco libero
- de-allocazione: restituzione del blocco aggiunta alla lista dei blocchi liberi

Problemi:

- le operazioni devono essere efficienti
- evitare lo spreco di memoria:
  - frammentazione interna
  - frammentazione esterna

Frammentazione: presente anche nella gestione memoria virtuale tramite **segmentazione**

- su uno spazio di memoria lineare
- vengono assegnati blocchi di dimensione variabile
- che, dopo un certo tempo, possono essere liberati

I due problemi sono analoghi e hanno soluzioni simili

# Frammentazione

- Frammentazione **interna**: lo spazio richiesto è  $X$ ,
  - viene allocato un blocco di dimensione  $Y > X$ ,
  - lo spazio  $Y-X$  è sprecato,
  - meno problematica della:
- Frammentazione **esterna**:
  - la continua allocazione e de-allocazione di blocchi porta alla creazione di blocchi liberi di piccole dimensioni
    - differenze tra il blocco libero usato e lo spazio effettivamente allocato
  - spazio sprecato per l'esistenza di piccoli blocchi difficilmente usabili, non è possibile allocare un blocco di grandi dimensioni, anche con tanta memoria libera

# Gestione della lista libera: unica lista

- Ad ogni richiesta di allocazione: cerca blocco di dimensione opportuna
  - **first fit**: primo blocco grande abbastanza
  - **best fit**: quello di dimensione più piccola, tra quelli sufficienti.
- Se il blocco scelto è abbastanza più grande di quello che serve, viene diviso in due e la parte inutilizzata è aggiunta alla LL
- Quando un blocco è de-allocato, viene restituito alla LL (se un blocco adiacente è libero, i due blocchi sono “fusi” in un unico blocco).



# Gestione lista libera: First fit o Best Fit ?

## Vantaggi - Svantaggi:

- First fit:
  - più veloce
  - occupazione di memoria peggiore
- Best fit:
  - più lento
  - occupazione di memoria migliore

# Gestione della lista libera: più liste

- Con unica LL, costo allocazione lineare nel numero di blocchi liberi
- liste libere multiple, migliori tempi di ricerca:
  - ogni lista contiene blocchi liberi di dimensione simile.
  - si prende il primo blocco disponibile nella lista opportuna

## Buddy system: n liste;

- la lista k-esima ha blocchi di dimensione  $2^k$
- se si desidera un blocco di dimensione  $2^j$ , e la lista relativa è vuota:
  - si cerca un blocco, nella lista successiva, di dimensione doppia, viene diviso in due parti,
  - se anche la lista successiva è vuota, la procedura si ripete ricorsivamente.
- quando un blocco di  $2^k$  è de-allocato, viene riunito alla sua altra metà (buddy - compagno), se disponibile

**Fibonacci heap** simile, ma i blocchi hanno dimensioni numeri di Fibonacci,

- maggiore scelta di dimensione, i numeri di Fibonacci crescono più lentamente delle potenze di 2
- minore frammentazione interna

# Esercizi

Considerato il seguente frammento di programma, con scope statico, mostrare la struttura dei record di attivazione, con i link statici e dinamici, (e display) dopo la sequenza di chiamate di funzione A(), D(), B(), C(), D().

```
void A()  
{ void B()  
  { void C()  
    { }  
  }  
void D()  
{ }  
}
```

# Esercizi

Mostrare l'evoluzione dello stack di attivazione del seguente codice nei diversi meccanismi di scope (statico,dinamico), ed di implementazione (catena statica, display, A-List, CRT)

```
int x = 1, y = 2;
void A (){
    int y = 3, z = 4;
    x++; y++, z++; }
void B () {
    int x = 5, z = 6;
    void C (){
        int x = 7, y = 8;
        A()
        x++; y++, z++; }
    C ();
    x++; y++, z++; }
B();
```

# Esercizi

Mostrare l'evoluzione dello stack di attivazione del seguente codice nei diversi meccanismi di scope, ed di implementazione

```
int x = 1, y = x;
void A () {
    void B() {
        int y = 2, z = 3;
        void C() {
            x++; y++; z++; }
        void D() {
            int x = 4, z = 5;
            C(); z++; }
        D(); C(); E();}
    void E () {
        x++; y++;}
    B(); E();}
A();
```