Fondamenti di Informatica

Allievi Automatici

A.A. 2015-16

Strutture Dati Dinamiche

Che cos’è una struttura dati

- Per **struttura dati** si intende comunemente
  - la rappresentazione dei dati e
  - le operazioni consentite su tali dati
- In linguaggi più evoluti del C esistono *librerie* di strutture dati
  - la Standard Template Library in C++
  - le Collection in Java
  - …

Strutture dati più comuni

- Lista concatenata (linked list)
- Pila (stack)
- Coda (queue)
- Insieme (set)
- Multi-insieme (multiset o bag)
- Mappa (map)
- Albero (tree)
- Grafo (graph)
- …

Lista Concatenata

- Una vecchia conoscenza, ormai!

```
typedef struct nodo {
  Tipo dato;
  struct nodo *next;
} Nodo;
typedef Nodo * lista;
```

L'abbiamo già trattata come TDA

Pila (o stack)

- È una struttura dati con accesso limitato all’elemento "in cima", che è quello inserito più recentemente (LIFO: last in, first out)
- Nomi standard delle operazioni:
  - Push
    - Inserimento in cima
  - Pop
    - prelievo e rimozione dell’elemento in cima, che è per definizione l'ultimo elemento inserito
  - Top o Peek
    - Prelievo senza rimozione (cioè "sola lettura") dell'elemento in cima

Applicazioni della pila

- Controllo di bilanciamento di simboli (parentesi, tag XML, blocchi C, …)
  - Per ogni simbolo di "apertura" si impila un segnaposto che sarà rimosso quando si incontra il simbolo duale di "chiusura"
    - Se il simbolo non corrisponde al segnaposto sulla pila, allora la sequenza non è bilanciata
    - Esempio: {[][{}][](){}]()}{[()]()}[{}] →ok {[}{] ()} → ko
- Implementazione di chiamate a funzione
  - Pila di sistema e record di attivazione
- Valutazione di espressioni aritmetiche

Pila (o stack): implementazione

- PILA (stack):
  - I nuovi nodi possono essere aggiunti e cancellati solo dalla cima (top) della pila
  - La base della pila è indicata da un puntatore a **NULL**
  - È una versione **vincolata** della lista concatenata
- *push*
  - Aggiunge un nuovo nodo alla cima della pila
  - Non restituisce nulla al chiamante (void)
- *pop*
  - Cancella un nodo dalla cima della pila
  - Memorizza il valore cancellato
  - Restituisce un valore booleano al chiamante
    - **true** se l'operazione ha avuto successo

```
typedef struct sNode {
        int data;
        struct sNode * nextPtr;
} StackNode;
typedef StackNode * StackNodePtr;

void push( StackNodePtr *, int );
int pop( StackNodePtr * );
int isEmpty( StackNodePtr );
void printStack( StackNodePtr );
void instructions( void );
```

```
int main() {
  StackNodePtr stackPtr = NULL;  /* punta alla base (= alla cima!) della pila  */
  int choice, value;
  do { instructions();
          scanf( "%d", &choice );
          switch ( choice ) {
               case 1: printf( "Enter an integer: " );    /* caso push */
                         scanf( "%d", &value );
                         push( &stackPtr, value );
                         printStack( stackPtr );                    break;
               case 2: if ( !isEmpty( stackPtr ) )          /* caso pop */
                            printf( "The popped value is %d.\n", pop( &stackPtr ) );
                         printStack( stackPtr );                    break;
               case 3: printf( "End of run.\n" );                  break;
               default: printf( "Invalid choice.\n\n" );           break;
          }
  } while ( choice != 3 );
   return 0;
}
```

```
void instructions( void ) {
      printf( "Enter choice:\n");
      printf( "1 to push a value on the stack\n");
      printf( "2 to pop a value off the stack\n");
      printf( "3 to end program\n\n> ");
}


void push( StackNodePtr *topPtr, int info ) {
     StackNodePtr newPtr;
     newPtr = (StackNodePtr) malloc( sizeof( StackNode ) );
     if ( newPtr != NULL ) {
          newPtr->data = info;
          newPtr->nextPtr = *topPtr;
          *topPtr = newPtr;
     } else
          printf( "%d not inserted. No memory available.\n", info );
}
```

```
int pop( StackNodePtr *topPtr ) {
        StackNodePtr tempPtr = *topPtr;
        int popValue = (*topPtr)->data;
        *topPtr = (*topPtr)->nextPtr;
        free( tempPtr );
        return popValue;
}

void printStack( StackNodePtr currentPtr ) {
        while ( currentPtr != NULL ) {
                printf( "%d --> ", currentPtr->data );
                currentPtr = currentPtr->nextPtr;
        }
        printf( "NULL\n\n" );
}

int isEmpty( StackNodePtr topPtr ) {
        return topPtr == NULL;
}
```

Coda (o queue)

- In una coda l'accesso è ristretto all'elemento inserito meno recentemente (FIFO)
- Le operazioni tipiche supportate dalla coda sono:
  - enqueue o offer
    - aggiunge un elemento in coda
  - dequeue o poll o remove
    - preleva e cancella l'elemento di testa
  - getFront o peek
    - preleva ma non cancella l'elemento di testa

Coda (o queue)

- Coda (queue):
  - *First-In, First-Out (FIFO).*
  - I nodi sono cancellati solo dalla *testa*
  - I nodi vengono inseriti solo dalla *coda*
- Operazioni di inserimento e cancellazione
  - *Accoda* (inserimento) e *de-accoda* (cancellazione)
- Utile nella programmazione di sistema:
  - Modella le code della stampante, pacchetti accodati nella rete, richieste di accesso a file condivisi …
  - In generale gestione semplice di turni per l'accesso a risorse condivise

```
typedef struct qNode {  int data;
                        struct qNode *nextPtr;  } QueueNode;
typedef QueueNode *QueueNodePtr;
```

- Idea! → usiamo **due** puntatori, uno al primo elemento e uno all'ultimo
  - Così aggiungiamo i nodi da una parte e li preleviamo dall'altra, senza mai dover scandire l'intera struttura per individuarne il fondo
  - Ovviamente è più comodo accodare (aggiungere) in fondo e de-accodare in testa
    - Facendo il contrario, deallocando non si potrebbe agevolmente ottenere il nuovo valore per il puntatore all'ultimo elemento, che sarebbe il precedente del nodo deallocato

```
void printQueue( QueueNodePtr );
int isEmpty( QueueNodePtr );
int dequeue( QueueNodePtr *, QueueNodePtr * );
void enqueue( QueueNodePtr *, QueueNodePtr *, int );
void instructions( void );
```

```
int main() {
  QueueNodePtr firstPtr = NULL, lastPtr = NULL;
  int choice, item;
  do { instructions(); scanf( "%d", &choice );
          switch( choice ) {
             case 1: printf( "Enter an integer: " ); scanf( "\n%d", &item );
                        enqueue(&firstPtr, &lastPtr, item ); printQueue( firstPtr ); break;
             case 2: if ( !isEmpty( firstPtr ) ) {
                           item = dequeue( &firstPtr, &lastPtr );
                           printf( "%d has been dequeued.\n", item );
                         }
                         printQueue( firstPtr );            break;
              case 3: printf( "End of run.\n" );          break;
              default: printf( "Invalid choice.\n\n" );  break;
        }
  } while ( choice != 3 );
  return 0;
}
```

```
void printQueue( QueueNodePtr currentPtr ) {
        while ( currentPtr != NULL ) {
                printf( "[%d]--->", currentPtr->data );
                currentPtr = currentPtr->nextPtr;
        }
        printf( "NULL\n\n" );
}

int isEmpty( QueueNodePtr firstPtr ) {
        return firstPtr == NULL;
}

int dequeue( QueueNodePtr *firstPtr, QueueNodePtr *lastPtr ) {
        QueueNodePtr tempPtr = *firstPtr;
        int value = ( *firstPtr )->data;
        *firstPtr = ( *firstPtr )->nextPtr;
        if ( *firstPtr == NULL )
                *lastPtr = NULL;
        free( tempPtr );
        return value;
}
```

```
void enqueue( QueueNodePtr *firstPtr, QueueNodePtr *lastPtr, int value ) {
        QueueNodePtr newPtr;
        newPtr = malloc( sizeof( QueueNode ) );
        if ( newPtr == NULL ) {
            printf( "%c not inserted. No memory available.\n", value);
            return;
        }
        newPtr->data = value;
        newPtr->nextPtr = NULL;
        if ( isEmpty( *firstPtr ) )
            *firstPtr = newPtr;
        else
            ( *lastPtr )->nextPtr = newPtr;
        *lastPtr = newPtr;
}

void instructions( void ) {
        printf ( "Enter your choice:\n   1 to add an item to the queue\n"
        printf ( "   2 to remove an item from the queue\n   3 to end\n\n> " );
}
```

Insieme (o set)

- È come la lista, ma con il vincolo di non ammettere valori duplicati
- L'ordine in cui appaiono gli elementi nella lista è trasparente all'utente
  - Necessariamente, poiché non è significativo
- Di solito, per motivi di convenienza, gli insiemi si realizzano tramite liste ordinate
  - Così si possono velocizzare alcune operazioni, come inserimento, rimozione, ...

Prototipi per l'insieme

- int add (Tipo item, Insieme * i);
  - aggiunge l'elemento dato all'insieme e restituisce un valore (booleano) che indica se l'operazione ha avuto successo
- int remove (Tipo item, Insieme * i);
  - rimuove l'elemento indicato dall'insieme e restituisce un valore (booleano) che indica se l'operazione ha avuto successo
- int contains (Tipo item, Insieme i);
  - verifica se l'elemento dato è presente nell'insieme (restituisce un valore booleano)

Albero binario

- Un albero binario è una struttura dati dinamica in cui i nodi sono connessi tramite “rami” ad altri nodi in modo che:
  - c'è un nodo di partenza (la “radice”)
  - ogni nodo (tranne la radice) è collegato ad uno e un solo nodo “padre”
  - ogni nodo è collegato al massimo a **due** altri nodi, detti “figli” (max due → *binario*)
  - i nodi che non hanno figli sono detti “foglie”

Struttura di un albero binario

```
typedef struct nodo {
  Tipo dato;
  struct nodo * left;
  struct nodo * right;
} Nodo;
typedef Nodo * tree;
```

STRUTTURA RICORSIVA

Un albero è un nodo da cui "spuntano" due... alberi!

(l'albero destro e l'albero sinistro)

Piccoli esercizi su alberi binari

NATURALMENTE in versione ricorsiva!

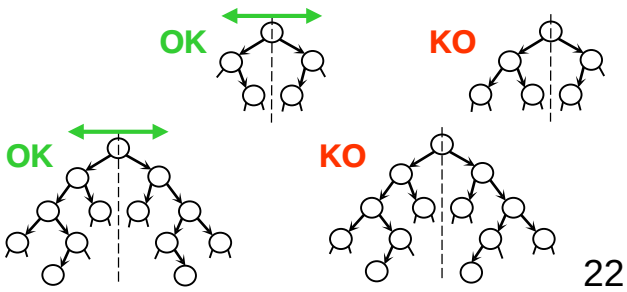
Per capire quanto sia più semplice formulare questi problemi e le relative soluzioni in forma ricorsiva è sufficiente … **provare** a fare **diversamente**!

Esercizi facili:

- Conteggio dei nodi
- Calcolo della profondità
- Ricerca di un elemento (in albero ordinato e non ordinato)
- Conteggio dei nodi foglia
- Conteggio dei nodi non-foglia
- Conteggio dei nodi su livello pari/dispari

Un esercizio (forse) un po' meno facile:

- Verifica di "simmetria speculare" dell'albero:



Conteggio dei nodi

```
int contaNodi ( tree t ) {
    if ( t == NULL )
        return 0;
    return     contaNodi( t->left ) +
               contaNodi( t->right ) +
               1  ;  /* c'è anche il nodo corrente */
}
```

Calcolo della profondità

```
int depth( tree t ) {
  int D, S;
  if ( t == NULL )
    return 0;
  S = depth( t->left );
  D = depth( t->right );
  if ( S > D )
      return S + 1;
  else
      return D + 1;
}
```

Ancora la profondità (variante)

```
int depth( tree t, int currentDepth ) {
    int D, S;
    if ( t == NULL )
        return currentDepth;
    S = depth( t->left, currentDepth+1 );
    D = depth( t->right, currentDepth+1 );
    if ( S > D )
        return S;
    else
        return D;
}
```

Questa versione utilizza il concetto "sussidiario" di *livello di profondità del nodo corrente*, che è incrementato di una unità ad ogni chiamata che scende "più in profondità"

Ricerca di un elemento in un albero (non ordinato)

Restituisce NULL se non trova nell'albero t il dato d, altrimenti restituisce il puntatore al primo nodo che lo contiene, effettuando la visita di t *in preordine sinistro*

```
tree trova( tree t, Tipo d) {
  tree temp;
  if ( t == NULL || t->dato == d ) /* Se Tipo ammette l'operatore == */
    return t;
  temp = trova( t->left, d );
  if ( temp == NULL )
    return trova( t->right, d );
  else
    return temp;
}
```

Ricerca di un elemento in un albero (ordinato)

Come il precedente, ma esplora sempre un solo ramo per ogni nodo – visita quindi al massimo depth(t) nodi prima di trovare il nodo cercato o decidere che non c'è

```
tree trova( tree t, Tipo d ) {
  if ( t == NULL || t->dato == d )      /* Se Tipo ammette l'operatore == */
    return t;

  if ( t->dato > d )                    /* Se Tipo ammette > e < */
    return trova( t->left, d );
  else
    return trova( t->right, d );
}
```

Gli altri sono lasciati per esercizio

Componibilità delle strutture dati

- Alberi di code
- Liste di liste
- Alberi di liste di code
- …
- Il “dato” contenuto nei nodi della struttura contenente è, in questi casi, una istanza di struttura dinamica del tipo che vogliamo sia il “contenuto”
  - del resto abbiamo sempre detto che il Tipo del contenuto dei nodi è assolutamente “libero”

Grafo

- Un grafo è un insieme di vertici (o nodi) collegati da lati (o archi)
- Può essere rappresentato, ad esempio, come la coppia dei due insiemi seguenti:
  - un insieme di vertici
  - un insieme di coppie di vertici (gli archi)

Albero n-ario

- È una generalizzazione dell'albero binario
  - ogni nodo ha un numero arbitrario di figli
- Può anche essere pensato come un grafo
  - Connesso
    - Cioè esiste un cammino di archi tra qualunque nodo e qualunque altro nodo nel grafo
  - Aciclico
  - Tale per cui
    - Ogni nodo (tranne uno, detto radice) ha un solo arco entrante
- Si usa ad esempio per rappresentare tassonomie e organizzazioni gerarchiche