UNIVERSITA' DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Elaborato per l'esame di
"Documentazione Automatica"
Prof. F.Cesarini

B+ Alberi

di Menchetti Sauro

A.A. 1997/98

Indice

Capitolo 1

Introduzione

Questo capitolo è dedicato ad una presentazione generale dei B+Alberi, delle loro principali caratteristiche e delle operazioni che si possono effettuare su di essi. Vengono anche introdotti i B Alberi e i B*Alberi che sono le strutture da cui derivano.

1.1 Caratteristiche generali

Il B+Albero è una delle principali strutture ad albero per la memorizzazione di indici per chiave primaria (un indice per chiave primaria è un insieme di coppie (k_i, r_i) ordinate in base alle chiavi k_i dove r_i è un riferimento alla registrazione con chiave k_i). E' adottato nella maggior parte dei sistemi per la gestione di basi di dati e in molti sistemi operativi, in particolare per le organizzazioni sequenziali con indice. Il suo principale pregio è quello di permettere la ricerca per intervalli di chiave mantenendo comunque limitato il numero di accessi alla memoria secondaria, cosa che invece non è possibile con i metodi hash (metodi che tramite un opportuno algoritmo di trasformazione della chiave, restituiscono l'indirizzo di una pagina dell'archivio in cui potrebbe trovarsi la registrazione). In generale viene utilizzato un indice a B+Albero per indirizzare le registrazioni dell'archivio; questo indice ha dimensioni tali da non poter essere tenuto, se non in piccola parte, in memoria centrale e quindi va organizzato opportunamente in memoria secondaria. I B+Alberi si prestano a questo in modo efficiente. Vediamo adesso come si arriva ai B+Alberi partendo dai B Alberi.

1.2 B+ Alberi come evoluzione dei B Alberi

1.2.1 B Alberi

Le principali caratteristiche di un B Albero sono le seguenti:

- è un albero perfettamente bilanciato (tutte le foglie stanno allo stesso livello) a più vie;
- ogni nodo dell'albero coincide con una pagina dell'archivio;
- detto *m* l'ordine dell'albero, ogni nodo diverso da una foglia ha al più *m* figli e *m – 1* chiavi;
- ogni nodo diverso da una foglia, tranne la radice, ha almeno $\lceil m / 2 \rceil$ figli e $\lceil m / 2 \rceil - 1$ chiavi;
- la radice ha almeno 2 figli e una chiave;
- ogni nodo non terminale con j + 1 figli contiene j chiavi;
- le chiavi sono ordinate nei nodi;
- l'altezza *h* di un B Albero è il numero di nodi che compaiono in un cammino dalla radice ad un nodo terminale;
- il costo dell'operazione di ricerca in un indice con *N* elementi è sempre minore od uguale dell'altezza *h* del B Albero, che cresce in modo logaritmico con *N*;
- l'utilizzo della memoria secondaria è almeno del 50%.

Ogni nodo del B Albero ha una struttura del tipo:

$$[\ p_0\ (k_1, r_1)\ p_1\ (k_2, r_2)\ p_2\ \ldots\ (k_j, r_j)\ p_j\ ]$$

dove j è il numero di chiavi del nodo, r_i è il riferimento al dato con chiave k_i e p_i è il riferimento al figlio (i p_i nelle foglie sono indefiniti). Inoltre p_0 punta al figlio con chiavi minori di k_1, p_i punta al figlio con chiavi maggiori di k_i ma minori di k_{i+1} e p_j punta al figlio con chiavi maggiori di k_j.

Altezza di un B Albero

Una caratteristica dei B Alberi è la possibilità di stimare facilmente la sua altezza *h* in funzione del numero di chiavi presenti; sia *N* il numero di chiavi presenti: allora vale la seguente relazione con l'altezza:

$$\log_m (N+1) \le h \le \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)$$

1.2.2 B*Alberi

I B*Alberi sono un'evoluzione dei B Alberi in cui tutte le coppie (k_j, r_j) sono memorizzate nei nodi terminali; inoltre tutte le chiavi più alte di ogni foglia sono riportate in una struttura a B Albero. I nodi terminali sono collegati a lista bidirezionale, ordinati per valore di chiave, e vengono chiamati *sequence set*. L'insieme dei nodi interni organizzato a B Albero, nei quali sono memorizzati solo chiavi e non riferimenti ai dati, è chiamato *index set*. Le chiavi memorizzate nei nodi interni servono solo per indirizzare la ricerca secondo il cammino che conduce al nodo terminale, nodo che potrebbe contenere la chiave cercata. I nodi terminali del B Albero hanno una struttura del tipo:

$$[\ p_p\ (k_1, r_1)\ ,\ (k_2, r_2)\ \ldots\ (k_m, r_m)\ p_s\ ]$$

dove p_p e p_s sono i riferimenti ai nodi precedente e successivo, mentre quelli intermedi hanno una struttura del tipo:

$$[\ p_0, k_1, p_1, k_2, p_2\ \ldots\ k_j, p_j\ ]$$

dove p_0 punta al figlio con chiavi minori o uguali di k_1, p_i punta al figlio con chiavi maggiori di k_i ma minori o uguali di k_{i+1} e p_j punta al figlio con chiavi maggiori di k_j.

1.2.3 B+ Alberi

I B+Alberi sono una variazione dei B*Alberi e rappresentano la struttura che generalmente viene usata per organizzare indici in memoria secondaria. I B+Alberi si ottengono dai B*Alberi modificando la struttura dei nodi intermedi che assumono la forma:

$$[\ k_1, p_1, k_2, p_2\ \ldots\ k_j, p_j\ ].$$

Ogni chiave k_i è la chiave più alta presente nel nodo puntato da p_i, sia esso foglia o nodo intermedio. L'aspetto delle foglie resta identico a quello delle foglie dei B*Alberi, mentre viene eliminato il primo puntatore dalle pagine interne. Inoltre viene introdotta una chiave fittizia ∞ che indica il massimo valore di chiave rappresentabile nell'archivio.

1.3 Operazioni sui B+ Alberi

Di seguito verranno illustrate le principali operazioni sui B+Alberi. Tali operazioni sono un'evoluzione di quelle dei B Alberi, in quanto devono considerare le caratteristiche aggiuntive dei B+Alberi. Si ipotizza di disporre di un'area di lavoro capace di contenere $h + 1$ nodi: in questo modo i nodi interessati nelle singole operazioni vengono trasferiti in memoria centrale una sola volta. Tali operazioni sono descritte in maggior dettaglio nella sezione dedicata all'illustrazione delle procedure.

1.3.1 Ricerca

L'operazione di ricerca, a differenza di quella di un B Albero, richiede sempre lo stesso numero di accessi in quanto i nodi intermedi servono per indirizzare la ricerca ed i riferimenti ai dati si trovano solo sui nodi terminali. La ricerca di una chiave *K* può avvenire iterativamente nel seguente modo:

1) Si definisce la radice come nodo corrente e si pone la variabile i a zero;
2) Finchè non siamo in un nodo terminale:
2.1) Finchè K è maggiore della i-esima chiave k_i del nodo corrente, incrementa i di 1;
2.2) Assegna al nodo corrente il nodo puntato da p_i;
3) Ricerca la chiave K nel nodo terminale puntato da p_i:
3.1) Se è presente, restituisci il numero della pagina dell'archivio in cui si trova;
3.2) Se non è presente, la ricerca termina senza successo.

Inoltre è opportuno salvare i nodi che costituiscono il cammino dalla radice alla foglia in cui termina la ricerca, in uno stack.

1.3.2 Inserimento

L'inserimento di una nuova chiave *K* richiede innanzitutto la ricerca della chiave stessa per vedere se è presente: in caso di ricerca senza successo, la nuova chiave viene inserita nell'ultimo nodo visitato che è un nodo terminale: l'albero così cresce dalle foglie verso la radice rimanendo sempre bilanciato, cioè mantenendo tutte le foglie allo stesso livello. La procedura di inserimento può essere schematizzata come segue:

1) Si definisce l'ultimo nodo memorizzato nello stack come nodo corrente;
2) Si assegna a Termine il valore falso;
3) Finchè Termine non è vero:
3.1) Se il nodo corrente non è pieno, si inserisce la chiave K nel nodo corrente e si assegna a Termine il valore vero;
3.2) Se il nodo corrente è pieno:
3.2.1) Se è la radice, si creano altri due nodi con cui dividere le chiavi e si assegna a Termine il valore vero;
3.2.2) Se non è la radice:
3.2.2.1) Se esiste un nodo adiacente non pieno, si ridistribuiscono con esso le chiavi e si assegna a Termine il valore vero;
3.2.2.2) Se tutti i nodi adiacenti sono pieni, si crea un nuovo nodo con cui suddividere le chiavi del nodo corrente e di un nodo adiacente pieno; si definisce il prossimo nodo dello stack come nodo corrente e si assegna alla chiave K la chiave più alta del nuovo nodo.

Come si vede, durante la procedura di inserimento di una chiave vengono utilizzate le informazioni salvate nello stack. L'operazione di ridistribuzione delle chiavi con un nodo adiacente non pieno è detta *bilanciamento,* mentre l'operazione di

suddivisione delle chiavi con un nodo adiacente pieno che porta alla creazione di un nuovo nodo, è detta *split.* L'utilizzazione minima della memoria secondaria aumenta dal 50% del B Albero al 67% del B+Albero. Il costo dell'operazione di inserimento, cioè il numero di accessi alla memoria secondaria, deve essere distinto a seconda che avvenga o meno la suddivisione di un nodo. Se non si rende necessaria la divisione, l'operazione richiede h letture e una riscrittura. In caso contrario, il caso peggiore è quello in cui la suddivisione si ripercuote fino alla radice inclusa, aumentando così di uno l'altezza dell'albero. Il costo dell'operazione in questo caso è di h letture e $2h + 1$ riscritture.

1.3.3 Cancellazione

Anche la cancellazione di una chiave K richiede innanzitutto la ricerca con successo della chiave stessa: il nodo da cui deve essere eliminata la chiave è l'ultimo nodo salvato sullo stack. Inoltre prima di cancellare una chiave bisogna vedere se la cancellazione fa scendere il numero di chiavi presenti nel nodo corrente al di sotto del limite inferiore ammesso. Quando si cancella una registrazione con chiave K, la coppia (chiave, riferimento) si rimuove dal nodo foglia; se la chiave K è usata come separatore nell'indice, non viene eliminata fintantoché i nodi per i quali serve come separatore non vengono fusi. La procedura di cancellazione può essere schematizzata come segue:

1) Si definisce l'ultimo nodo memorizzato nello stack come nodo corrente;

2) Si assegna a Termine il valore falso;

3) Finchè Termine non è vero:

3.1) Se siamo sopra il limite inferiore di chiavi ammesse, si cancella la chiave K dal nodo e si assegna a Termine il valore vero;

3.2) Se si scende sotto il limite inferiore:

3.2.1) Se il nodo corrente è la radice: se le chiavi nella radice sono solo due, può avvenire una cancellazione della radice a favore dell'unico figlio rimasto che diventa quindi la nuova radice, oppure una cancellazione dell'archivio per assenza di chiavi a seconda se la radice è intermedia o terminale; se le chiavi sono più di due, avviene una semplice cancellazione della chiave K; si assegna a Termine il valore vero;

3.2.2) Se il nodo corrente non è la radice:

3.2.2.1) Se esiste un nodo adiacente con un numero di chiavi al di sopra del limite inferiore, si ridistribuiscono con esso le chiavi e si assegna a Termine il valore vero;

3.2.2.2) Se tutti i nodi adiacenti contengono proprio un numero di chiavi uguale al limite inferiore, si fonde il nodo corrente con un nodo adiacente provocando la cancellazione di una pagina; si definisce il prossimo nodo dello stack come nodo corrente e si aggiorna la chiave K.

Anche nell'operazione di cancellazione vengono utilizzate le informazioni salvate nello stack. La procedura che porta alla eliminazione di un nodo è detta *concatenazione*, mentre la procedura di ridistribuzione con un nodo adiacente è detta *bilanciamento*. Il costo dell'operazione di inserimento va da un minimo di h letture e una riscrittura quando la chiave appartiene ad un nodo terminale con un numero di chiavi al di sopra del limite inferiore ammesso, ad un massimo di $2h - 1$ letture e $h + 1$ riscritture quando la cancellazione si ripercuote fino alla radice.

1.3.4 Ricerca per intervallo di chiavi

Una delle ragioni che porta a preferire un'organizzazione a B+Albero rispetto ad una basata su metodi hash, è la possibilità di effettuare ricerche per valori della chiave in un intervallo. L'intervallo però non deve essere troppo ampio, altrimenti non conviene usare l'indice per la ricerca, ma è meglio procedere con una scansione delle pagine dei dati per trovare le registrazioni che soddisfano la condizione.

Capitolo 2

Descrizione del programma

Questo capitolo descrive in tutti i suoi particolari la struttura del programma, le variabili in gioco e le procedure. Il programma è scritto in linguaggio C utilizzando lo standard ANSI C. Per la compilazione e la generazione del file eseguibile è stato usato un compilatore Borland C++ versione 3.1 del 1992.

2.1 Struttura del programma

Il programma è suddiviso in varie procedure in modo ma mantenere la suddivisione concettuale del problema. Tutte le procedure sono costituite da un numero limitato di linee: ho preferito creare procedure anche di poche linee pur di aumentare la chiarezza del programma. Gli identificatori cercano di illustrare in modo chiaro qual è il significato della variabile o della procedura a cui sono associati.

2.1.1 Le parti del programma

Sono tre le parti in cui può essere suddiviso il programma: una prima in cui sono introdotte le principali strutture dati e definite la maggior parte delle costanti, una seconda che contiene il programma principale ed una terza che raccoglie tutte le procedure. Le variabili, le strutture dati, le procedure e i punti di maggiore rilevanza sono commentati nel listato del programma. Il programma consente di creare un nuovo archivio oppure di utilizzarne uno già creato in precedenza; dopo una qualsiasi operazione, consente di verificare il corretto funzionamento delle operazioni eseguite stampando sullo schermo tutte le informazioni utili allo scopo. Inoltre durante l'esecuzione appaiono sullo schermo dei commenti che descrivono quello che sta accadendo.

2.1.2 I files del programma

Vengono utilizzati tre files per realizzare l'archivio con indice a B+Albero: uno è il file primario con l'indice a B+Albero, l'altro è il file secondario che contiene l'archivio vero e proprio e l'ultimo contiene delle informazioni sui due precedenti files indispensabili per la loro gestione: tale file viene chiamato file delle informazioni. Tutti e tre i files sono binari: il file dell'indice e quello dell'archivio sono paginati, mentre quello delle informazioni non è paginato (paginare un file significa suddividere il file in pagine: una pagina rappresenta l'unità di trasferimento tra programma e sistema di archiviazione e la sua dimensione è stabilita dal programma con un'opportuna dichiarazione della struttura della registrazione dell'archivio). Il file delle informazioni viene ogni volta letto e scritto per intero, mentre gli altri due vengono letti e scritti per pagine. I files dell'indice e dell'archivio sono ad accesso diretto: tramite un riferimento si accede direttamente alla pagina desiderata. Il riferimento è semplicemente il numero della pagina all'interno del file: le pagine sono numerate in ordine crescente a partire da uno. La politica di inserimento cancellazione esegue delle cancellazioni logiche con recupero dello spazio, cioè una pagina od un dato non sono mai cancellati fisicamente dall'archivio ma vengono semplicemente etichettati ed inseriti in una lista, pronti per un'eventuale successivo utilizzo. Quando occorre una nuova pagina nel file dell'indice, se la lista delle pagine vuote non contiene nessuna pagina, la nuova pagina viene inserita in fondo al file, altrimenti viene utilizzata la pagina libera con numero d'ordine minore appartenente alla lista. Per quanto riguarda gli inserimenti e le cancellazioni nel file dell'archivio, esiste una lista che contiene le pagine in cui sono avvenute delle cancellazioni: quando deve essere inserita una nuova registrazione, se non ci sono pagine che hanno subito cancellazioni, tale registrazione viene inserita nell'ultima pagina non piena, altrimenti viene scelta la pagina con numero d'ordine minore appartenente alla lista.

2.1.3 L'ordine delle procedure

Non tutte le procedure sono visibili al programma principale che può utilizzare solo quelle più importanti come la ricerca, l'inserimento, la cancellazione e la modifica di una chiave. Le procedure sono disposte in un ordine particolare, rispettando le regole di visibilità: le prime possono essere utilizzate da quelle che seguono, mentre le ultime possono essere richiamate solo se esplicitamente dichiarate all'interno di quelle precedenti. Quindi una procedura non utilizza le procedure seguenti, ma solo quelle precedenti per portare a termine il suo compito. L'unica che utilizza delle procedure definite nelle parti successive del programma è il `main()`, mentre tutte le altre usano solo quelle definite precedentemente. Così, ad esempio, la procedura di inserimento è preceduta da tutte quelle procedure che sono utili a portare a termine l'operazione: nessuna procedura successiva a quella di inserimento è da essa utilizzata. L'ordine in cui sono definite le procedure è lo stesso in cui sono dichiarate nel programma principale. Le variabili passate ad una procedura sono chiamate parametri. La definizione di alcune strutture dati di limitato utilizzo come i buffer per l'inserimento, non avviene all'inizio del programma ma nel punto dove verranno poi utilizzate. Le variabili all'interno delle procedure vengono generalmente definite solo quando sono necessarie e non prima. Si è utilizzata una sola variabile globale in modo da rendere di utilizzo più generale le varie procedure.

2.1.4 La gestione degli errori

Sono gestiti, per quanto possibile, anche la maggior parte degli errori che si possono verificare durante l'esecuzione del programma: se il programma termina correttamente, il `main()` restituisce all'ambiente il valore di ritorno zero a significare una corretta terminazione. Nelle procedure di salvataggio e caricamento dei dati, vengono gestiti gli errori di pagina non positiva (le pagine sono numerate in modo crescente a partire da uno), il fatto di non poter aprire il file, eventuali errori di posizionamento, di scrittura e di lettura. Gli errori fatali sono gestiti con degli `exit()`, che restituiscono all'ambiente valori di ritorno diversi da zero a segnalare una terminazione anomala del programma: ogni `exit()` restituisce un numero diverso da tutti gli altri. Ogni volta che si richiede memoria per poter allocare un puntatore, si verifica prima se c'è spazio sufficiente per poter eseguire l'operazione. Inoltre ogni puntatore viene deallocato quando non serve più. Durante il programma sono definiti diversi vettori: se, a causa dell'inserimento di molte registrazioni nell'archivio, le costanti che indicano il numero di elementi dei vettori sono troppo piccole, viene comunicato all'esterno quale costante deve essere modificata. Ad esempio, nella procedura di stampa dell'albero, può avvenire che i due vettori di `MAXLIVELLO` elementi siano troppo piccoli per contenere tutti i nodi utili alla stampa. Nelle schermate di scelta tra più alternative, se si preme un numero che non corrisponde a nessuna scelta, il programma emette un bip e permette poi di inserire un nuovo numero. Tutti gli errori sono commentati con delle frasi che cercano di illustrare che tipo di errore è avvenuto.

2.2 Le costanti

Le costanti sono definite all'inizio del programma subito dopo le inclusioni delle varie librerie; altre costanti si trovano poi lungo il programma. Sono caratterizzate dal fatto di essere scritte in maiuscolo, per poter essere distinte meglio dagli altri identificatori. In questa sezione descriveremo le costanti all'inizio del programma. Riportiamo il listato dove sono definite tali costanti:

```
#define   ORDINE_M          5
#define   NUM_DATI          6

#define   INFINITO   1000000L

#define   NODO_INTERMEDIO   0
#define   NODO_TERMINALE    1

#define   FILE_INDICE       "indice.dat"
#define   FILE_ARCHIVIO     "archivio.dat"
#define   FILE_INFORMAZIONI "informaz.dat"

#define   LIMITE_INF        (ORDINE_M + 1) / 2

#define   MAX_CASUALE    1000
```

La costante ORDINE_M indica l'ordine del B+Albero, cioè il numero di chiavi e quindi di riferimenti presenti nel nodo. La costante NUM_DATI indica il numero di registrazioni, cioè di record, memorizzabili in una pagina dell'archivio. Le costanti NODO_INTERMEDIO e NODO_TERMINALE sono usate nella struttura dati che memorizza i nodi dell'albero e servono a distinguere un nodo terminale da uno intermedio. Gli identificatori FILE_INDICE, FILE_ARCHIVIO e FILE_INFORMAZIONI sostituiscono delle stringhe costanti che non sono altro che i nomi dei tre files in cui sono registrati rispettivamente l'indice, l'archivio e le informazioni. La costante LIMITE_INF indica il numero minimo di chiavi che può contenere un nodo: in un B+Albero tale limite è fissato. Infine la costante MAX_CASUALE indica il massimo valore di una chiave generata casualmente.

2.3 Le strutture dati

Le principali strutture dati sono dichiarate esternamente a qualsiasi procedura, in modo da poter essere utilizzate in qualsiasi punto che segue la dichiarazione. Sono descritti i nodi dell'albero e le pagine dell'archivio insieme alla procedura di inizializzazione, la struttura che contiene le informazioni sui files, lo stack con le sue procedure di push() e pop() e i buffer per la procedura di inserimento.

2.3.1 I nodi del B+ Albero

Riportiamo di seguito la parte di programma in cui è definita tale struttura:

```
typedef long int Tnumpagina;
typedef long int Tchiave;
typedef int Boolean;

typedef struct nodo *Tpuntnodo;

typedef struct chiavi_puntatori {
 Tchiave     chiave;
 Tnumpagina  nptr;
 } Tchiavi_puntatori[ORDINE_M];

typedef struct nodo {
 Tnumpagina         np;
 Boolean            int_ter;
 Tnumpagina         npp;
 Tnumpagina         nps;
 Tchiavi_puntatori  info;
 } Tnodo;
```

Prima di creare la struttura dati per i nodi dell'albero, vengono definiti alcuni tipi di dato utili per la struttura stessa ed anche in seguito: questo serve ad aumentare la chiarezza e la leggibilità del programma. Gli identificatori di tipo hanno la prima lettera maiuscola. Il tipo Tnumpagina rappresenta il numero della pagina

memorizzata nel file dell'indice o in quello dell'archivio. Il tipo `Tchiave` è il tipo della chiave (si assume che tutte le chiavi dell'archivio siano non negative), mentre il tipo `Boolean` indica che la variabile è una variabile booleana, cioè che può assumere solo i due valori vero e falso. Il tipo `Tpuntnodo` è il puntatore ai nodi dell'albero. La struttura `chiavi_puntatori`, ridefinita poi con il nome `Tchiavi_puntatori`, contiene quella parte del nodo che memorizza le chiavi ed i riferimenti alle pagine (riferimenti ad altri nodi nel caso di nodo intermedio, riferimenti alle pagine dell'archivio in cui si trova la registrazione nel caso di nodo terminale). Il tipo `Tnodo`, che è la struttura per memorizzare i nodi dell'albero, è un record e può essere rappresentato graficamente come segue:

altre informazioni				chiavi e riferimenti				
np	int_ter	npp	nps	chiave	nptr	…	chiave	nptr

Il campo `np` contiene la pagina del file dell'indice in cui è memorizzato il nodo, il campo `int_ter` indica se il nodo è intermedio oppure terminale (si fa uso delle costanti `NODO_INTERMEDIO` e `NODO_TERMINALE`), i campi `npp` e `nps` hanno senso solo se il nodo è terminale ed indicano la pagina del nodo precedente e del nodo successivo nella catena di nodi del sequence set. Infine il campo `info` contiene le chiavi e i riferimenti alle pagine.

Inizializzazione di un nodo

Quando occorre usare un nuovo nodo, è necessario prima inizializzarlo, in modo che le procedure che lo utilizzano possano partire tutte dalla stessa configurazione iniziale. Ecco la procedura che si occupa di fare questo:

```
void inizializza_nodo(Tpuntnodo p)
{
 p->np  = 0L;
 p->int_ter = NODO_TERMINALE;
 p->npp = 0L;
 p->nps = 0L;
 for (int i = 0; i < ORDINE_M; ++i)
 {
       p->info[i].chiave = -1L;
       p->info[i].nptr   =  0L;
 }
}
```

Inizialmente il nodo viene etichettato come terminale in quanto, quando un nuovo nodo viene aggiunto all'albero, esso assume la posizione di nodo terminale. I campi che indicano il numero di una pagina vengono messi a zero: zero non è un valore ambiguo in quanto le pagine di un file sono numerate in ordine crescente a partire da uno. Le chiavi sono invece inizializzate a meno uno in quanto si ipotizza che tutte le chiavi presenti nell'archivio siano non negative. Si può dunque dire che il valore zero per una pagina e il valore meno uno per una chiave indicano che

quel campo è ancora inutilizzato; anche durante la fase di cancellazione, per cancellare una chiave od un riferimento ad una pagina, basta assegnargli i rispettivi valori iniziali meno uno e zero. Infine, poiché un nodo del B+Albero coincide con una pagina del file dell'indice, la struttura di un nodo è la stessa di quella di una pagina dell'indice.

2.3.2 Le pagine dell'archivio

Subito dopo la definizione della struttura dati per i nodi, viene introdotta la struttura dati per le pagine dell'archivio:

```
typedef struct dato {
 Tchiave chiave;
 char cognome[20];
 char nome[20];
 } Tdato;

typedef Tdato Tpagina[NUM_DATI];

typedef Tpagina *Tpuntpagina;
```

La registrazione memorizzata in una pagina dell'archivio è costituita dalla chiave, da un nome e da un cognome: le chiavi vengono sempre riportate nell'archivio per sicurezza e completezza e poter poi verificare la correttezza delle operazioni avvenute. Il tipo `Tpagina` rappresenta la pagina dell'archivio che non è altro che un vettore di `NUM_DATI` record di tipo `Tdato`; infine viene definito il tipo puntatore ad una pagina `Tpuntpagina`.

Inizializzazione di una pagina

Quando occorre aggiungere una nuova pagina all'archivio, ecco come viene inizializzata:

```
void inizializza_pagina(Tpuntpagina p)
{
 for (int i = 0; i < NUM_DATI; ++i)
 {
       (*p)[i].chiave  = -1L;
       strcpy((*p)[i].cognome, "");
       strcpy((*p)[i].nome,    "");
 }
}
```

Il valore della chiave è inizializzato a meno uno, mentre ai campi cognome e nome viene assegnata la stringa vuota; quando una chiave deve essere cancellata da

una pagina, viene marcata con meno uno ed i campi cognome e nome con la stringa vuota.

2.3.3 Le informazioni sui files

Ecco la struttura che contiene le informazioni sui files dell'indice e dell'archivio:

```
#define  MAXDEL    100

typedef Tnumpagina Tpagdel[MAXDEL];

typedef struct info_file_indice {
 Tnumpagina pagina_radice;
 Tnumpagina pagina_sequence_set;
 Tnumpagina pagina_vuota_infondo;
 Tpagdel pagine_cancellate;
 } Tinfo_file_indice;

typedef struct info_file_archivo {
 Tnumpagina pagina_nonpiena_ultima;
 Tpagdel pagine_cancellazioni;
 } Tinfo_file_archivio;

typedef struct info_file_indice_archivio {
 Tinfo_file_indice indice;
 Tinfo_file_archivio archivio;
 } Tinfo_file_indice_archivio;
```

Come prima cosa, possiamo subito notare la definizione di una nuova costante `MAXDEL`, che serve a costruire un vettore di riferimenti a pagine chiamato `Tpagdel`: questo vettore serve a memorizzare le pagine dell'indice cancellate oppure le pagine dell'archivio che hanno subito cancellazioni. Le informazioni sul file dell'indice e sul file dell'archivio sono definite in due strutture separate `Tinfo_file_indice` e `Tinfo_file_archivio`, ma poi sono raccolte nell'unica struttura chiamata `Tinfo_file_indice_archivio`. Per quanto riguarda le informazioni sul file dell'indice, il campo `pagina_radice` indica in quale pagina si trova la radice dell'albero, informazione indispensabile per varie procedure come la ricerca e la stampa dell'albero, il campo `pagina_sequence_set` indica qual è la prima pagina del sequence set, il campo `pagina_vuota_infondo` contiene il numero dell'ultima pagina vuota in fondo al file dell'indice e infine il vettore `pagine_cancellate` contiene tutte le pagine che vengono cancellate dalla procedura di concatenazione. Per quanto riguarda le informazioni sul file dell'archivio vero e proprio, il campo `pagina_nonpiena_ultima` indica qual è l'ultima pagina dell'archivio non completamente piena, mentre il vettore `pagine_cancellazioni` contiene tutte le pagine che hanno subito delle cancellazioni.

L'unica variabile globale del programma

La definizione:

```
Tinfo_file_indice_archivio informazioni;
```

definisce l'unica variabile globale del programma chiamata `informazioni`: essendo una variabile esterna, essa è inizializzata automaticamente a zero in ogni suo campo. Tale variabile contiene tutte le informazioni utili sui files dell'indice e dell'archivio ed è visibile in ogni punto del programma.

2.3.4 Lo stack

Lo stack è definito più avanti nel programma in modo da rimanere invisibile alle procedure che non lo utilizzano; viene creato durante la fase di ricerca di una chiave e memorizza tutti i nodi che costituiscono il cammino dalla radice alla foglia in cui termina la ricerca: quest'informazione è utile alle procedure di inserimento e di cancellazione in caso di propagazione ai livelli superiori.

```
#define MAXVAL 20

int sp;
Tnumpagina stack[MAXVAL];
```

La costante `MAXVAL` indica la massima profondità dello stack, la variabile `sp` è lo stack pointer, cioè quella variabile che punta alla prossima posizione libera dello stack e infine il vettore `stack` contiene i riferimenti alle pagine dell'indice. Essendo variabili esterne, sono inizializzate automaticamente a zero.

Le procedure che gestiscono lo stack: push e pop

Ecco le due uniche procedure che manipolano i dati dello stack:

```
void push(Tnumpagina pag)
{
 if (sp < MAXVAL)
       stack[sp++] = pag;
 else
 {
       printf("\n Errore: lo stack è pieno");
       printf("\n Aumentare MAXVAL");
       getch();
 }
}
```

```
Tnumpagina pop(void)
{
 if (sp > 0)
       return stack[--sp];
 else
 {
       printf("\n Errore: lo stack è vuoto");
       getch();
       return 0L;
 }
}
```

La procedura `push()` inserisce il riferimento ad una pagina `pag` in cima allo stack ed incrementa di uno lo stack pointer, mentre la procedura `pop()` decrementa di uno lo stack pointer e ritorna il valore in cima allo stack. Entrambe le procedure gestiscono le condizioni di errore di stack pieno e stack vuoto.

2.3.5 I buffer per la procedura di inserimento

Quando l'inserimento di una chiave avviene in un nodo pieno, occorrono delle strutture temporanee in grado di contenere tutte le chiavi ed i riferimenti alle pagine che non entrerebbero in un solo nodo: tali strutture sono chiamate buffer. Ecco le dichiarazioni nelle linee che seguono:

```
typedef struct buffer {
 Tchiave chiave;
 Tnumpagina nptr;
 } Vet[ORDINE_M + 1];

typedef struct buffer_rid {
 Tchiave chiave;
 Tnumpagina nptr;
 } Vet_rid[2 * ORDINE_M];

typedef struct buffer_adiac {
 Tchiave chiave;
 Tnumpagina nptr;
 } Vet_adiac[2 * ORDINE_M + 1];
```

Ogni buffer è un vettore di record con due campi: il campo `chiave` e il campo `nptr` che è il riferimento alle pagine. La dimensione del buffer varia a seconda della procedura: se si tratta di un inserimento nella radice piena è `ORDINE_M + 1`, se invece si deve fare ridistribuzione con un nodo adiacente è `2*ORDINE_M` e infine se si deve fare lo split è `2*ORDINE_M + 1`. I vettori `Vet`, `Vet_rid` e `Vet_adiac` sono rispettivamente il buffer per l'inserimento nella radice piena, per l'inserimento con ridistribuzione e per l'inserimento con split.

La procedura di inserimento nel buffer

La procedura di inserimento nel buffer inserisce le chiavi in modo ordinato: il buffer non deve essere vuoto quando inizia l'inserimento ma deve contenere già delle chiavi. I parametri della procedura sono la chiave da inserire, il riferimento alla pagina, il buffer e la dimensione del buffer (la dimensione del buffer in ingresso rende più generale la procedura).

```
void inserisci_ord_nel_buffer(Tchiave chiave, Vet v,
                              Tnumpagina n, int dimbuf)
{
 int i = 0;
 while ((chiave > v[i].chiave) && (v[i].chiave != -1L))
       ++i;
 int posizione = i;
 for (i = dimbuf - 2; i >= posizione; --i)
 {
       v[i + 1].chiave = v[i].chiave;
       v[i + 1].nptr   = v[i].nptr;
 }
 v[posizione].chiave = chiave;
 v[posizione].nptr   = n;
}
```

La procedura trova la posizione in cui va inserita la chiave che viene salvata nella variabile `posizione`; viene creato un posto libero facendo scorrere a destra di una posizione tutte le chiavi maggiori di quella che deve essere inserita ed infine la nuova chiave viene inserita nella giusta posizione.

2.4 Il programma principale

Si trova subito dopo la dichiarazione della struttura che memorizza le informazioni sui files: si suddivide in due parti e permette di gestire le varie operazioni effettuabili sull'archivio.

2.4.1 La dichiarazione delle procedure

Nella prima parte vengono dichiarate le procedure che saranno utilizzate dal programma principale. Ci sono le procedure di salvataggio, caricamento e stampa delle informazioni sui files, di un nodo e di una pagina, la ricerca, l'inserimento, la cancellazione e la modifica di una chiave, la procedura di presentazione, la procedura di stampa del B+Albero ed infine la procedura che genera chiavi casuali. L'ordine in cui sono dichiarate è lo stesso in cui sono definite.

2.4.2 Il nucleo del programma principale

Dopo la pagina di presentazione, il programma permette di creare un nuovo archivio oppure di utilizzarne uno già creato; se si decide di creare un nuovo archivio, eventuali files con lo stesso nome di quelli che gestisce il programma vengono cancellati, mentre se si decide di utilizzare un archivio già presente, vengono caricate e stampate le informazioni sui files dell'indice e dell'archivio. La successiva schermata permette di scegliere tra varie operazioni come indicano le seguenti linee di programma:

```
printf("\n Premere il numero corrispondente \n\n");
printf("\n 1) Inserimento casuale di varie chiavi");
printf("\n 2) Inserimento di una chiave");
printf("\n 3) Cancellazione di una chiave");
printf("\n 4) Modifica di una chiave");
printf("\n 5) Ricerca di una chiave");
```

Il nucleo del programma principale è uno `switch` che gestisce le precedenti operazioni. L'inserimento casuale di varie chiavi fissa il cognome e il nome da inserire, chiede quante chiavi devono essere inserite e memorizza tale valore nella variabile `chiavi_casuali`; tramite la procedura di inserimento `inserisci_chiave()`, inserisce nell'archivio le chiavi generate casualmente nel seguente modo:

```
Tchiave chiave_casuale(void)
{
 time_t t;

 srand((unsigned) time(&t));
 return (rand() % MAX_CASUALE);
}
```

La procedura `chiave_casuale()` restituisce una chiave casuale compresa tra zero e `MAX_CASUALE - 1`. Per verificare il corretto funzionamento del programma, viene stampato il B+Albero dalla radice alle foglie insieme a tutte le pagine dell'archivio e dell'indice. L'operazione di inserimento di una chiave chiede qual è la chiave da inserire insieme al cognome e al nome, la cancellazione chiede la chiave da cancellare: alla fine di entrambe le procedure avviene la stampa dell'albero e di tutte le pagine dell'indice e dell'archivio. La modifica di una chiave richiede la chiave da modificare insieme al nuovo cognome e nome, mentre la procedura di ricerca restituisce il nome ed il cognome associati ad una data chiave. Le procedure di inserimento e di cancellazione stampano anche sullo schermo delle frasi che descrivono quello che sta avvenendo. Alla fine di ognuna di queste operazioni, vengono stampate e salvate le informazioni sui files. L'ultima schermata del programma chiede se si vogliono effettuare altre operazioni.

2.5 Le procedure

In questo paragrafo verranno illustrate tutte le procedure che fanno parte del programma, con particolare riguardo verso le due procedure d'inserimento e di cancellazione che sono le più complesse.

2.5.1 Le procedure di stampa

Le procedure di stampa, come quelle di salvataggio e cancellazione, si suddividono in tre gruppi: quelle che riguardano le informazioni sui files, quelle che riguardano i nodi dell'albero e quelle che riguardano le pagine dell'archivio. Per le pagine ed i nodi esistono anche due procedure di stampa diretta da file. Riportiamo la dichiarazione di tali procedure senza il listato:

```
void stampa_informazioni(Tinfo_file_indice_archivio inf);
void stampa_nodo(Tpuntnodo punt);
void stampa_pagina(Tpagina pag);

void stampa_nodo_file(Tnumpagina num);
void stampa_pagina_file(Tnumpagina num);
```

Le procedure di stampa non fanno altro che prendere i vari campi delle strutture e stamparli sullo schermo; le due procedure di stampa da file prendono in ingresso il numero della pagina da stampare invece che il puntatore alla struttura.

2.5.2 Le procedure di salvataggio

Sono tre le procedure di salvataggio: una riguarda le informazioni sui files, una i nodi e l'altra le pagine:

```
void salva_informazioni(Tinfo_file_indice_archivio inf);
void salva_nodo(Tpuntnodo punt, Tnumpagina num);
void salva_pagina(Tpuntpagina pg, Tnumpagina num);
```

La procedura di salvataggio delle informazioni è una semplice procedura di scrittura su file binario che riscrive completamente ogni volta il file delle informazioni. La procedura di salvataggio di un nodo e di una pagina operano per pagine e richiedono anche il posizionamento sul file che viene fatto tramite la variabile `num` e la funzione `fseek()`: queste ultime due procedure rilasciano, dopo il salvataggio, la memoria allocata per il rispettivo puntatore.

2.5.3 Le procedure di caricamento

Riportiamo la dichiarazione delle procedure di caricamento:

```
void carica_informazioni(Tinfo_file_indice_archivio
                                                    *pinf);
Tpuntnodo carica_nodo(Tnumpagina num);
Tpuntpagina carica_pagina(Tnumpagina num);
```

La procedura di caricamento delle informazioni è una semplice procedura di lettura da file binario che legge ogni volta l'intero file delle informazioni, mentre il caricamento di un nodo e di una pagina operano per pagine e richiedono prima l'allocazione della memoria per il puntatore, poi un posizionamento su file (fatto tramite la variabile `num` e la funzione `fseek()`); restituiscono entrambe un puntatore alla corrispondente struttura.

2.5.4 Le procedure che gestiscono le informazioni sui files

Sono cinque le procedure che tengono aggiornate le informazioni sui files: tre legate alla procedura di inserimento e due legate alla procedura di cancellazione. Ecco la dichiarazione di tali procedure:

```
Tnumpagina prima_pagina_archivio(void);
Tnumpagina prima_pagina_indice(void);
void verifica_pagina_piena(Tnumpagina nppa);

void informazioni_archivio_cancellazioni(Tnumpagina pag);
void informazioni_indice_cancellate(Tnumpagina pag);
```

La procedura `prima_pagina_archivio()` restituisce il numero della prima pagina non piena dell'archivio dove può essere inserita una chiave: tale pagina viene scelta come quella con numero minore tra l'ultima pagina non piena dell'archivio e le pagine che hanno subito delle cancellazioni; la variabile `npg` memorizza il numero di tale pagina. La procedura `prima_pagina_indice()` restituisce il numero della prima pagina vuota dell'indice, utile quando deve essere allocato un nuovo nodo: tale pagina viene scelta come quella con numero minore tra l'ultima pagina vuota dell'indice e le pagine completamente cancellate in seguito a concatenazioni. La variabile `npg` memorizza il numero di tale pagina. Inoltre vengono aggiornate anche le informazioni sui files, cosa che invece la procedura `prima_pagina_archivio()` lascia fare alla procedura `verifica_pagina_piena()`: se `npg` coincide con l'ultima pagina vuota dell'indice, tale valore viene incrementato di uno e si aggiunge una nuova pagina al file dell'indice inizializzata nel modo corretto; altrimenti, se `npg` coincide con una delle pagine facenti parte della lista delle pagine cancellate, si pone tale valore a zero. La procedura `verifica_pagina_piena()` verifica se dopo l'inserimento dell'ultima registrazione in una pagina dell'archivio, tale pagina risulta piena (questo viene fatto tramite la procedura `check_pagina_piena()` che scandisce tutta la pagina che gli viene passata come parametro alla ricerca di

un posto libero): se la pagina è diventata piena, vengono aggiornate le informazioni sui files come nella procedura `prima_pagina_indice()`.
Le procedure `informazioni_archivio_cancellazioni()` e `informazioni_indice_cancellate()` si occupano di inserire nella prima posizione libera dei rispettivi vettori, il numero delle pagine che hanno subito delle cancellazioni e delle pagine che sono state completamente cancellate: tale numero è memorizzato nel parametro `pag`.

2.5.5 La procedura di ricerca

La procedura di ricerca realizza l'algoritmo illustrato nella sezione dedicata alle operazioni sui B+Alberi ed è costituita da due diverse procedure: una prima che trova il nodo terminale in cui potrebbe esserci la chiave ed una seconda che esegue una ricerca binaria nel nodo terminale per vedere se la chiave è presente. Esaminiamo adesso la prima: essa restituisce meno uno se l'archivio è ancora vuoto, zero se la chiave non è presente e il numero della pagina dell'archivio in cui si trova la chiave se essa è presente; vengono anche salvati sullo stack i nodi che formano il cammino dalla radice alla foglia in cui è terminata la ricerca. I parametri della procedura di ricerca sono la variabile `chiave` e la variabile `pag_radice` che contiene il numero della pagina in cui si trova la radice dell'albero. La variabile interna `pp` punta al nodo corrente mentre la variabile `indice` contiene il risultato della ricerca binaria eseguita su un nodo terminale: meno uno se la chiave non è presente, un numero compreso tra zero e `ORDINE_M - 1` se la chiave è presente. La variabile `temp` è una variabile temporanea e serve per poter rilasciare l'area di memoria puntata da `pp`. Per prima cosa si inizializza lo stack pointer `sp` a zero, poi si esamina se l'archivio è vuoto. Se non lo è, finché non siamo in un nodo terminale, si scandisce il nodo corrente fino a quando non si trova una chiave più alta di quella da ricercare; la variabile `i` contiene la posizione di tale chiave. Il nodo corrente viene sostituito con il nodo puntato da `pp->info[i].nptr`, preoccupandosi anche di salvare nello stack il nodo che è stato sostituito e di rilasciare il puntatore allocato. Una volta giunti in un nodo terminale, si effettua una ricerca binaria della chiave: è possibile fare questo in quanto le chiavi sono ordinate nei nodi. La procedura di ricerca binaria restituisce la posizione del nodo in cui si trova la chiave oppure meno uno se la chiave non è presente e viene fatta solo su un nodo terminale:

```
typedef int Indice_nodo;

Indice_nodo ricerca_binaria(Tchiave chiave, Tchiave v[],
                                              int num_chiavi)
{
 Indice_nodo low, high, mid;

 low = 0;
 high = num_chiavi - 1;
 while (low <= high)
 {
       mid = (low + high ) / 2;
```

```
        if (chiave < v[mid])
              high = mid - 1;
        else if (chiave > v[mid])
              low = mid + 1;
        else
              return mid;
    }
    return -1;
}
```

Il tipo `Indice_nodo` è l'indice della chiave all'interno di un nodo e va da zero a `ORDINE_M - 1`: indica la posizione della chiave nel nodo. I parametri della procedura di ricerca binaria sono un buffer di chiavi `v[]`, il numero di chiavi `num_chiavi` su cui si deve fare la ricerca e la chiave da ricercare, dati che le vengono passati dalla procedura di ricerca vera e propria.

2.5.6 La procedura di inserimento

L'operazione di inserimento è molto complessa e si suddivide in molte piccole procedure. L'algoritmo su cui si basa è quello descritto nel capitolo precedente. Iniziamo a descrivere la procedura che viene invocata ogni volta che c'è da fare un inserimento; descriveremo poi le varie procedure utili a portare a termine l'inserimento ma che non sono visibili al programma principale. Caratteristica di tutte le procedure che inseriscono una chiave è il fatto di dover distinguere se il nodo è intermedio o terminale: nel caso di nodo terminale, la registrazione deve essere memorizzata in una pagina del file dell'archivio. La procedura d'inserimento viene dichiarata all'interno del `main()` con la seguente linea:

```
void inserisci_chiave(Tchiave chiave, Tnumpagina radice,
                              char cog[20], char nome[20]);
```

I parametri sono la chiave, il numero della pagina della radice, il cognome e il nome da inserire. Come prima cosa viene eseguita la ricerca della chiave facendo uso della variabile `radice`: la variabile `npag` contiene il risultato della ricerca. Quindi si verifica se l'archivio è vuoto: se lo è, viene creato inserendovi la prima chiave. Sono così allocati e inizializzati il puntatore alla pagina `ppag` e il puntatore al nodo `pnodo`, vengono creati i files dell'indice e dell'archivio dalla procedura `crea_files()`; il nodo e la pagina appena inizializzati sono aggiornati con le nuove informazioni, le informazioni sui files vengono modificate nel seguente modo:

```
informazioni.indice.pagina_radice            = 1L;
informazioni.indice.pagina_sequence_set      = 1L;
informazioni.indice.pagina_vuota_infondo     = 2L;
informazioni.archivio.pagina_nonpiena_ultima = 1L;
```

Il nuovo nodo e la nuova pagina vengono salvati negli opportuni files. In fondo al file dell'indice viene salvato anche un altro nodo subito dopo essere stato inizializzato.

Se l'archivio non è vuoto, si verifica se la chiave è già presente: se lo è, non si esegue nessuna operazione, altrimenti si passa alla procedura di inserimento in un archivio già creato. Tale procedura ha la seguente dichiarazione:

```
void inserimento(Tchiave chiave, char cognome[20],
                                          char nome[20]);
```

I parametri sono gli stessi della precedente procedura ad esclusione del numero di pagina della radice. La variabile interna `pp` punta al nodo in cui deve essere inserita la chiave (che è quello che si trova in cima allo stack), la variabile `nnpag` serve in caso di propagazione ai livelli superiori causata da uno split e la variabile booleana `termine` inizializzata a zero, indica quando la procedura d'inserimento deve terminare. Si inizia testando se il nodo puntato da `pp` è pieno: un nodo è pieno quando `pp->info[ORDINE_M - 1].chiave` è diverso da meno uno, in quanto le chiavi vengono inserite in modo ordinato da sinistra a destra.

Se il nodo non è pieno, la procedura `inserisci_nodo_non_pieno()` si occupa di portare a terminare l'inserimento, distinguendo il caso di nodo intermedio o terminale. Vediamone il listato:

```
void inserisci_nodo_non_pieno(Tchiave c, char cog[20],
        char nom[20], Tpuntnodo puntnodo, Tnumpagina nnp)
{
 Tnumpagina nppla;
 Tpuntpagina ppag;
 if (puntnodo->int_ter)
 {
       nppla = prima_pagina_archivio();
       ppag = carica_pagina(nppla);
 }

 inserisci_ord_nel_nodo(c, puntnodo, (puntnodo->int_ter)
                                          ? nppla : nnp);

 if (puntnodo->int_ter)
 {
       inserisci_nella_pagina(c, cog, nom, *ppag);
       salva_pagina(ppag, nppla);
       verifica_pagina_piena(nppla);
 }
 salva_nodo(puntnodo, puntnodo->np);
}
```

Le variabili interne `nppla` e `ppag` servono nel caso in cui il nodo sia terminale: la prima contiene il numero della prima pagina non piena dell'archivio, la seconda è il puntatore a tale pagina. Il parametro `nnp` serve nel caso in cui il nodo non sia terminale e contiene il numero della pagina del figlio che deve essere inserito nel nodo insieme alla chiave. Queste tre variabili sono presenti in ogni procedura di inserimento. Viene fatto uso delle procedure `inserisci_ord_nel_nodo()` che inserisce ordinatamente la chiave in un nodo e `inserisci_nella_pagina()` che inserisce la registrazione nella prima posizione libera della pagina dell'archivio. La prima opera nello stesso modo della procedura di inserimento ordinato in un buffer, la seconda scandisce la pagina dell'archivio finché non trova una posizione libera. L'espressione condizionale:

```
(puntnodo->int_ter) ? nppla : nnp;
```

determina quale variabile deve essere usata come parametro per la procedura `inserisci_ord_nel_nodo()`: `nppla` se il nodo è terminale oppure `nnp` se non lo è.

Se il nodo è pieno, si verifica se tale nodo è la radice: nel caso che sia la radice, la procedura `inserisci_radice_piena()` termina l'inserimento, distinguendo il caso di radice intermedia o terminale. Il parametro `pt` punta alla radice piena. Tale procedura utilizza un buffer di appoggio `vet` dove vengono copiate tutte le chiavi e i riferimenti alle pagine della radice: in tale buffer viene anche inserita la nuova chiave insieme al suo riferimento. A questo punto vengono creati due nuovi nodi: le variabili interne `pp1` e `pp2` puntano rispettivamente al nodo figlio ed alla nuova radice. Le chiavi vengono divise a metà tra il nodo puntato da `pt` e quello puntato da `pp1` e nella radice `pp2` viene inserita la chiave più alta di `pt`. La procedura prosegue aggiornando tutte le informazioni dei nodi e quelle sui files; infine tutte le modifiche apportate vengono salvate nei rispettivi files.

Se il nodo corrente non è la radice, si verifica se è possibile eseguire una ridistribuzione delle chiavi con un nodo adiacente non pieno tramite la procedura `ridistribuzione_ins()`. La dichiarazione di tale procedura è la seguente:

```
Tpuntnodo ridistribuzione_ins(Tpuntnodo pp);
```

Il parametro `pp` punta al nodo in cui deve essere inserita la chiave; la procedura restituisce il puntatore ad un nodo non pieno adiacente a `pp` oppure `NULL` se tale nodo non esiste. La variabile interna `padre` punta al nodo padre di `pp`. La linea:

```
Indice_nodo indice = cerca(padre, pp->np);
```

assegna alla variabile `indice` il valore ritornato dalla procedura `cerca()`. Quest'ultima procedura cerca nel padre il riferimento al figlio e restituisce l'indice di tale riferimento. Se `indice` è uguale a zero, si esamina il nodo immediatamente a destra di `pp`, mentre se `indice` è uguale a `ORDINE_M - 1`, si esamina il nodo immediatamente a sinistra di `pp`: se il nodo esaminato puntato dalla variabile `pfra` non è pieno, si restituisce `pfra` altrimenti `NULL`. Se invece `indice` è maggiore di zero e minore di `ORDINE_M - 1`, si esaminano entrambi i nodi adiacenti a `pp` puntati da `pfra1` e `pfra2`: se uno dei due non è pieno, si restituisce il puntatore a quel nodo altrimenti `NULL`. Nel caso siano entrambi non pieni, viene restituito in modo casuale uno dei due puntatori.

Se si può fare la ridistribuzione delle chiavi con un nodo adiacente non pieno, cioè se la variabile interna `pfratello` della procedura `inserimento()` è diversa da `NULL`, la procedura `inserisci_ridistribuzione()` termina l'inserimento. Viene utilizzato il buffer `vet` per contenere le chiavi dei due nodi. Innanzitutto il nodo puntato da `puntnodo` viene copiato nel buffer, poi vengono inserite ordinatamente nel buffer tutte le chiavi di `pfratello` e infine anche la chiave da inserire viene immessa nel buffer. La variabile `destra` indica se il nodo si trova a destra o a sinistra. Sfruttando quest'informazione, le chiavi immesse del buffer sono ripartite tra `pfratello` e `puntnodo` ed anche il nodo `padre` viene modificato; si aggiornano tutte le informazioni sui nodi e sui files ed il tutto viene salvato in memoria secondaria.

Infine, se tutti i nodi adiacenti sono pieni, la procedura `inserisci_split()` esegue l'inserimento della chiave creando le condizioni per la propagazione ai livelli superiori. I parametri della procedura non sono più delle semplici variabili, ma i loro indirizzi: questo perché la procedura deve modificare il contenuto dei parametri (in altre parole le variabili sono passate per indirizzo e non per valore). La dichiarazione della procedura è la seguente:

```
void inserisci_split(Tchiave *c, char cog[20],
     char nom[20], Tpuntnodo *puntnodo, Tnumpagina *nnp);
```

La procedura `fratello_adiacente()` restituisce un nodo adiacente alla variabile che gli viene passata come argomento; opera nello stesso modo della procedura `ridistribuzione_ins()`, soltanto che non testa se il nodo è pieno. La linea:

```
Tpuntnodo pfratello = fratello_adiacente(*puntnodo);
```

assegna alla variabile interna `pfratello` un nodo adiacente a `*puntnodo`. La variabile `vet` è il buffer che serve per fare lo split: qui vengono inserite tutte le chiavi ed i riferimenti di `*puntnodo` e di `pfratello`. Anche la chiave da inserire viene immessa nel buffer. Le due linee:

```
Tnumpagina npp1 = prima_pagina_indice();
Tpuntnodo pp1 = carica_nodo(npp1);
```

assegnano alla variabile `pp1` il nuovo nodo che serve per completare lo split. Le costanti `primo` e `secondo` indicano in che modo devono essere ripartite le chiavi del buffer fra i tre nodi `*puntnodo`, `pfratello` e `pp1`; la variabile `destra` indica se `pfratello` si trova a destra o a sinistra di `*puntnodo`. Sfruttando queste informazioni, le chiavi vengono ripartite fra i tre nodi, tenendo presente un'eventuale modifica del nodo padre. Tutte le informazioni sui nodi e sui files sono aggiornate e salvate in memoria secondaria. Le tre linee:

```
(*c) = pp1->info[secondo - primo - 1].chiave;
(*nnp) = pp1->np;
(*puntnodo) = carica_nodo(pop());
```

aggiornano le tre variabili passate per indirizzo che servono per propagare l'inserimento ai livelli superiori: in questo modo vengono sfruttate le informazioni contenute nello stack. L'operazione di inserimento riparte dalla linea:

```
while (!termine)
{
        ...
}
```

della procedura `inserimento()`.

Di seguito è riportato l'elenco completo delle procedure che eseguono l'operazione di inserimento nell'ordine in cui compaiono nel programma:

```
void crea_files(void);

void inserisci_ord_nel_nodo(Tchiave chiave,
                            Tpuntnodo pn, Tnumpagina n);

void inserisci_nella_pagina(Tchiave chiave,
               char cog[20], char nome[20], Tpagina pag);

Indice_nodo cerca(Tpuntnodo padre, Tnumpagina np);

Tpuntnodo ridistribuzione_ins(Tpuntnodo pp);

Tpuntnodo fratello_adiacente(Tpuntnodo pp);
```

```
void inserisci_nodo_non_pieno(Tchiave c, char cog[20],
       char nom[20], Tpuntnodo puntnodo, Tnumpagina nnp);

void inserisci_radice_piena(Tchiave c, char cog[20],
             char nom[20], Tpuntnodo pt, Tnumpagina nnp);

void inserisci_ridistribuzione(Tchiave c, char cog[20],
       char nom[20],Tpuntnodo puntnodo, Tnumpagina nnp,
                                      Tpuntnodo pfratello);

void inserisci_split(Tchiave *c, char cog[20], char
          nom[20], Tpuntnodo *puntnodo, Tnumpagina *nnp);

void inserimento(Tchiave chiave, char cognome[20],
                                            char nome[20]);

void inserisci_chiave(Tchiave chiave, Tnumpagina radice,
                             char cog[20], char nome[20]);
```

2.5.7 La procedura di cancellazione

Anche l'operazione di cancellazione è molto complessa e come quella di inserimento si suddivide in molte piccole procedure. L'algoritmo su cui si basa è quello illustrato nel capitolo precedente. La struttura della procedura di cancellazione è la stessa di quella di inserimento: ogni procedura che fa parte dell'operazione di inserimento ha la sua controparte nell'operazione di cancellazione. La procedura che viene invocata ogni volta che c'è da cancellare una registrazione è dichiarata all'interno del `main()` nel seguente modo:

```
void cancella_chiave(Tchiave chiave,
                                    Tnumpagina pag_radice);
```

Facendo uso del parametro `pag_radice`, si esegue una ricerca della chiave e si assegna il risultato alla variabile interna `npagarc`: se l'archivio è vuoto oppure se la chiave non è presente, non si esegue alcuna operazione.

Se la chiave è presente, viene attivata la procedura di cancellazione in un archivio già creato che ha la seguente dichiarazione:

```
void cancellazione(Tchiave chiave, Tnumpagina npagar);
```

Il parametro `npagar` indica il numero della pagina dell'archivio in cui si trova la registrazione che deve essere cancellata nel caso in cui il nodo sia terminale, la variabile interna `pp` punta al nodo in cui deve essere cancellata la chiave e la variabile `termine` indica quando deve terminare la procedura di cancellazione. Viene fatto uso della procedura `conta_chiavi_nodo()` che conta il numero di

chiavi presenti in un nodo: anche il valore `INFINITO` viene contato come se fosse una chiave.

Se il numero di chiavi nel nodo corrente puntato da `pp` è al di sopra del limite inferiore di chiavi ammesse in un nodo, la procedura `cancella_sopra_limite()` termina l'operazione di cancellazione. La procedura `cancella_dal_nodo()` cancella dal nodo passato come parametro la chiave indicata: finché il parametro `chiave` non è maggiore della i-esima chiave del nodo, si incrementa di uno la variabile `i`; nella variabile `posizione` viene memorizzato l'indice della chiave che deve essere cancellata. Tutte le chiavi più alte di quella da cancellare vengono fatte scorrere a sinistra di una posizione e si assegna il valore meno uno all'ultima chiave rimasta nel nodo. Si testa poi se il nodo puntato dalla variabile interna `punt` è intermedio o terminale: se è intermedio, l'unica operazione che viene eseguita è quella di salvataggio del nodo che ha subito la cancellazione; altrimenti, se il nodo è terminale, viene caricata la pagina dell'archivio che contiene la registrazione nella variabile `ppag`: la procedura `cancella_dalla_pagina()` scandisce tutte le chiavi della pagina fino a quando non trova la chiave da cancellare che viene così eliminata. Infine la pagina dell'archivio modificata viene salvata e vengono aggiornate le informazioni sui files riguardanti le cancellazioni.

Se il numero di chiavi nel nodo corrente scende sotto il limite inferiore ammesso a seguito della cancellazione, si esamina se tale nodo è la radice: se lo è, la procedura `cancella_dalla_radice()` termina l'operazione di cancellazione. Per prima cosa, si assegna alla variabile interna `nchiavi` il numero di chiavi presenti nella radice individuata dal parametro `pradice`. Si esamina poi quanto tale variabile: se è uguale a due, si distinguono due casi: quello di radice intermedia e quello di radice terminale. Se la radice è terminale, la cancellazione della registrazione provoca lo svuotamento dell'archivio: tutti e tre i files del programma vengono riportati nella condizione di archivio vuoto. Se la radice è intermedia, viene cancellata a favore dell'unico figlio rimasto individuato dal numero di pagina `pradice->info[1].nptr`. Se le chiavi sono più di due, avviene una semplice cancellazione, distinguendo sempre i due casi di radice intermedia o terminale.

Se il nodo corrente non è la radice, si verifica se è possibile eseguire una ridistribuzione delle chiavi con un nodo adiacente che contiene un numero di chiavi superiore al limite inferiore. La procedura `ridistribuzione_can()` si occupa di fare tale verifica e di assegnare il risultato alla variabile `pfrat`. Opera nello stesso modo della procedura `ridistribuzione_ins()`, ad eccezione per il fatto che verifica se il numero di chiavi del nodo adiacente è al di sopra del limite inferiore anziché testare se il nodo è pieno. La dichiarazione di tale procedura è la seguente:

```
Tpuntnodo ridistribuzione_can(Tpuntnodo pp);
```

Se la variabile `pfrat` è diversa da `NULL`, la procedura `cancella_ridistribuzione()` termina l'operazione di cancellazione. La variabile

ppadre punta al padre del nodo corrente. La procedura cancella_dal_nodo() elimina la chiave dal nodo puntato da pp. Si inizializza poi il buffer vet in cui vengono copiate le chiavi di pp e di pfratello; la variabile destra indica se il nodo adiacente a quello corrente si trova a destra o a sinistra. Sfruttando queste informazioni, le chiavi vengono ripartite tra i due nodi pp e pfratello ed anche il nodo ppadre viene aggiornato. Se la variabile destra è diversa da zero, le prime LIMITE_INF - 1 chiavi del buffer vet vengono copiate in pp mentre le restanti sono assegnate al nodo pfratello; se la variabile destra è nulla, avviene il procedimento inverso. La procedura termina aggiornando tutte le informazioni e salvando le modifiche apportate nei rispettivi files, preoccupandosi eventualmente di cancellare la registrazione dall'archivio.

Infine, se tutti i nodi adiacenti contengono un numero di chiavi uguale al limite inferiore ammesso, la procedura cancella_concatenazione() esegue la cancellazione della chiave creando le condizioni per la propagazione ai livelli superiori. I parametri della procedura non sono più delle semplici variabili, ma i loro indirizzi come avviene nella procedura inserisci_split(). La dichiarazione della procedura è la seguente:

```
void cancella_concatenazione(Tchiave *chiave,
                        Tpuntnodo *pp, Tnumpagina npagar);
```

La procedura fratello_adiacente() restituisce un nodo adiacente alla variabile che gli viene passata come argomento; opera nello stesso modo della procedura ridistribuzione_can(), soltanto che non testa se il numero di chiavi del nodo è al di sopra del limite inferiore ammesso. La linea:

```
Tpuntnodo pfratello = fratello_adiacente(*pp);
```

assegna alla variabile interna pfratello un nodo adiacente a *pp. La variabile ppadre punta al padre del nodo corrente e la variabile destra indica in quale posizione si trova il nodo adiacente. Come prima cosa, si cancella la chiave dal nodo corrente; poi, se la variabile destra è diversa da zero, si copia *pp in pfratello, altrimenti è pfratello che viene copiato in *pp: questo viene fatto per gestire in modo semplice il valore INFINITO. Inoltre se il nodo corrente è terminale, la registrazione viene cancellata dall'archivio. Il nodo rimasto inutilizzato viene inizializzato e salvato nel file dell'indice; tutte le informazioni sui files vengono aggiornate. In particolare si può notare l'aggiornamento della catena di nodi che costituiscono il sequence set: se il nodo cancellato non è né il primo né l'ultimo della catena, si provvede a modificare le informazioni del nodo adiacente a quest'ultimo diverso da quello con cui sono state concatenate le chiavi. Le linee:

```
if (destra)
```

```
{
      (*chiave) = ppadre->info[cerca(ppadre,
                                     (*pp)->np)].chiave;
      ...
}
else
{
      (*chiave) = ppadre->info[cerca(ppadre,
                                  pfratello->np)].chiave;
      ...
}
(*pp) = carica_nodo(pop());
```

aggiornano le variabili passate per indirizzo che servono per propagare la cancellazione ai livelli superiori. L'operazione di cancellazione riparte dalla linea:

```
while (!termine)
{
       ...
}
```

della procedura `cancellazione()`.

Di seguito è riportato l'elenco completo delle procedure usate per realizzare l'operazione di cancellazione nell'ordine in cui compaiono nel programma:

```
int conta_chiavi_nodo(Tpuntnodo punt);

void cancella_dal_nodo(Tchiave chiave, Tpuntnodo pp);

void cancella_dalla_pagina(Tchiave chiave,
                                    Tpuntpagina puntpag);

Tpuntnodo ridistribuzione_can(Tpuntnodo pp);

void cancella_sopra_limite(Tchiave chiave,
                      Tpuntnodo punt, Tnumpagina npagar);

void cancella_dalla_radice(Tchiave chiave,
                   Tpuntnodo pradice, Tnumpagina npagarc);

void cancella_ridistribuzione(Tchiave chiave,
   Tpuntnodo pp, Tnumpagina npagar, Tpuntnodo pfratello);

void cancella_concatenazione(Tchiave *chiave,
                       Tpuntnodo *pp, Tnumpagina npagar);

void cancellazione(Tchiave chiave, Tnumpagina npagar);

void cancella_chiave(Tchiave chiave,
                                    Tnumpagina pag_radice);
```

2.5.8 La procedura di modifica

La procedura di modifica dell'informazione contenuta in una registrazione dell'archivio esegue, per prima cosa, una ricerca della chiave: se l'archivio è vuoto o la chiave non è presente, non viene effettuata alcuna operazione. Se la chiave è presente, viene caricata la pagina che la contiene; tale pagina viene scandita finché non si trova la registrazione da modificare; per ultimo il nuovo cognome ed il nuovo nome sostituisco quelli presenti e la pagina così corretta viene salvata. La dichiarazione di tale procedura è la seguente:

```
void modifica_dati(Tchiave chiave, Tnumpagina radice,
                                   char c[20], char n[20]);
```

Il parametro `chiave` indica qual è la registrazione da modificare, il parametro `radice` indica in quale pagina si trova la radice e serve per ricercare la chiave, infine i vettori di caratteri `n` e `c` contengono il nuovo nome e cognome. La variabile interna `npagarc` memorizza, in caso di ricerca con successo, il numero della pagina in cui si trova la chiave, mentre la variabile `ppag` punta alla pagina dell'archivio con la registrazione da modificare.

2.5.9 La procedura di stampa dell'albero

La procedura di stampa dell'albero conclude la panoramica sulle procedure. E' costituita da quattro procedure: una prima che inizializza a zero il vettore passato come parametro, una seconda che copia il contenuto di un vettore in un altro, un'altra che stampa le chiavi di un nodo e l'ultima che gestisce la stampa dell'albero. L'albero viene stampato per livelli da sinistra a destra partendo dalla radice: un nodo è separato da un altro tramite appositi separatori. Quando un livello non entra più in una linea dello schermo, il nodo che causa questo problema viene spezzato in due e si riprende dalla linea successiva; due new line separano un livello da un altro. Non ci sono parametri in ingresso alla procedura: l'unica informazione che serve è contenuta nella variabile globale `informazioni` ed è il numero della pagina in cui si trova la radice nel file dell'indice. La procedura di stampa gestisce due vettori `figli_1_` e `figli_2_` di `MAXLIVELLO` riferimenti a pagine. Il vettore `figli_1_` viene inizializzato a zero e nella sua prima posizione viene copiato il riferimento alla radice dell'albero; una variabile booleana `finito` viene posta falsa. Finchè la variabile `finito` non diventa vera, si inizializza a zero il vettore `figli_2_` e si scandisce il vettore `figli_1_`: se l'i-esimo elemento è diverso da zero, viene caricato il nodo corrispondente nella variabile interna `pnodo` (che viene deallocata quando non serve più) e vengono stampate le chiavi di tale nodo tramite la procedura `stampa_chiavi_nodo()`; se il nodo è intermedio, vengono copiati in `filgli_2_` i riferimenti alle pagine dei figli altrimenti, se il nodo è terminale, viene assegnato il valore vero alla variabile `finito`. Infine, quando il vettore `figli_1_` è stato completamente scandito, il vettore `figli_2_` viene copiato in `figli_1`.

Listato del programma

```
/*****************************************************************************

                                    B+ ALBERI

*****************************************************************************/

#include  <conio.h>
#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <time.h>

#define   ORDINE_M          5   // ordine del B+Albero: numero di chiavi in un nodo
#define   NUM_DATI          6   // numero di record in una pagina di FILE_ARCHIVIO

#define   INFINITO   1000000L   // massimo valore della chiave nell'archivio

#define   NODO_INTERMEDIO   0
#define   NODO_TERMINALE    1

#define   FILE_INDICE       "indice.dat"          // file che contiene l'indice
#define   FILE_ARCHIVIO     "archivio.dat"        // file che contiene l'archivio
#define   FILE_INFORMAZIONI "informaz.dat"        // file che contiene le
                                                  // informazioni sui files

#define   LIMITE_INF        (ORDINE_M + 1) / 2    // numero minimo di chiavi
                                                  // memorizzabili in un nodo

#define   MAX_CASUALE    1000                     // massimo valore della chiave
                                                  // generata casualmente

/***************************** DEFINIZIONE DEI TIPI ******************************/

typedef long int Tnumpagina;          // tipo che indica il numero della pagina

typedef long int Tchiave;             // tipo della chiave

typedef int Boolean;                  // 1 significa vero, 0 significa falso

typedef struct nodo *Tpuntnodo;       // tipo puntatore ai nodi del B+Albero

typedef struct chiavi_puntatori {     // tipo informazione del nodo
       Tchiave     chiave;            // chiave numerica
       Tnumpagina  nptr;              // riferimento ad una pagina
       } Tchiavi_puntatori[ORDINE_M];

typedef struct nodo {                 // tipo nodo del B+Albero
       Tnumpagina         np;         // numero della pagina del nodo in FILE_INDICE
       Boolean            int_ter;    // indica se è un nodo intermedio o terminale
       Tnumpagina         npp;        // numero della pagina del nodo precedente
       Tnumpagina         nps;        // numero della pagina del nodo successivo
       Tchiavi_puntatori  info;       // informazione del nodo
       } Tnodo;

typedef struct dato {                 // dati memorizzati in FILE_ARCHIVIO
       Tchiave chiave;
       char cognome[20];
       char nome[20];
       } Tdato;

typedef Tdato Tpagina[NUM_DATI];      // tipo pagina dell'archivio

typedef Tpagina *Tpuntpagina;         // puntatore alla pagina dell'archivio


/*

Dichiarazione della struttura che contiene le informazioni
sul file dell'indice e sul file dell'archivio

*/
```

```
#define  MAXDEL    100        // indica un limite superiore per le cancellazioni

typedef Tnumpagina Tpagdel[MAXDEL];

typedef struct info_file_indice {          // informazioni FILE_INDICE
       Tnumpagina pagina_radice;           // pagina in cui si trova la radice
       Tnumpagina pagina_sequence_set;     // 1° pagina del sequence set
       Tnumpagina pagina_vuota_infondo;    // 1° pagina vuota in fondo all'indice
       Tpagdel pagine_cancellate;          // insieme delle pagine cancellate
       } Tinfo_file_indice;

typedef struct info_file_archivo {         // informazioni FILE_ARCHIVIO
       Tnumpagina pagina_nonpiena_ultima;  // ultima pagina non piena dell'archivio
       Tpagdel pagine_cancellazioni;       // pagine con delle cancellazioni e
       } Tinfo_file_archivio;              // che quindi non sono del tutto piene

typedef struct info_file_indice_archivio { // struttura completa delle informazioni
       Tinfo_file_indice indice;           // sui due files
       Tinfo_file_archivio archivio;
       } Tinfo_file_indice_archivio;

/******************************** VARIABILE GLOBALE ********************************/

// variabile globale esterna (inizializzata tutta a zero) che contiene
// le informazioni sui files FILE_INDICE e FILE_ARCHIVIO

Tinfo_file_indice_archivio informazioni;

/************************* INIZIO PROGRAMMA PRINCIPALE ***************************/

int main(void)
{
       void presentazione(void);
       Tchiave chiave_casuale(void);

       void stampa_informazioni(Tinfo_file_indice_archivio info);
       void salva_informazioni(Tinfo_file_indice_archivio info);
       void carica_informazioni(Tinfo_file_indice_archivio *pinfo);

       void salva_nodo(Tpuntnodo punt, Tnumpagina num);
       Tpuntnodo carica_nodo(Tnumpagina num);
       void stampa_nodo(Tpuntnodo punt);
       void stampa_nodo_file(Tnumpagina num);

       void salva_pagina(Tpuntpagina ppag, Tnumpagina num);
       Tpuntpagina carica_pagina(Tnumpagina num);
       void stampa_pagina(Tpagina pag);
       void stampa_pagina_file(Tnumpagina num);

       Tnumpagina ricerca_chiave(Tchiave chiave, Tnumpagina radice);
       void inserisci_chiave(Tchiave chiave, Tnumpagina radice,
                                          char cognome[20], char nome[20]);
       void cancella_chiave(Tchiave chiave, Tnumpagina radice);
       void modifica_dati(Tchiave chiave, Tnumpagina radice,
                                          char cognome[20], char nome[20]);

       void stampa_albero(void);

       presentazione();

       char r;
       do
       {

       clrscr();
       printf("\n Premere il numero corrispondente\n\n");
       printf("\n 1) Creazione di un nuovo archivio");
       printf("\n 2) Utilizzo dell'archivio già presente");
       gotoxy(80, 25);

       char q;
       do
       {
              q = getch();
```

```
        if (q != '1' && q != '2')
                printf("\a");
} while (q != '1' && q != '2');

clrscr();
printf("\n Premere il numero corrispondente\n\n");
printf("\n 1) Inserimento casuale di varie chiavi");
printf("\n 2) Inserimento di una chiave");
printf("\n 3) Cancellazione di una chiave");
printf("\n 4) Modifica di una chiave");
printf("\n 5) Ricerca di una chiave");
gotoxy(80, 25);

char c;
do
{
        c = getch();
        if (c != '1' && c != '2' && c != '3' && c != '4' && c != '5')
                printf("\a");
} while (c != '1' && c != '2' && c != '3' && c != '4' && c != '5');

if (q == '1') // nuovo archivio
{
        remove(FILE_INFORMAZIONI);
        remove(FILE_INDICE);
        remove(FILE_ARCHIVIO);
}
else // vecchio archivio
{
        carica_informazioni(&informazioni);
        clrscr();
        stampa_informazioni(informazioni);
        getch();
}

switch (c)
{
        case '1': // inserimento casuale

        clrscr();
        printf("\n\n\n INSERIMENTO CASUALE DI VARIE CHIAVI\n");

        char cog[20] = "Menchetti";
        char nom[20] = "Sauro";

        int chiavi_casuali;
        printf("\n Immetti il numero di chiavi da inserire: ");
        scanf("%ld", &chiavi_casuali);

        for (int i = 0; i < chiavi_casuali; ++i)
        {
                inserisci_chiave(chiave_casuale(),
                                informazioni.indice.pagina_radice, cog, nom);

                stampa_albero();

                clrscr();
                printf("\n Vuoi vedere un riepilogo");
                printf(" di tutte le pagine (s/n) ? ");

                char w;
                do
                {
                        w = getch();
                        if (w != 's' && w != 'n')
                                printf("\a");
                } while (w != 's' && w != 'n');

                if (w == 's')
                {

                clrscr();
                printf("\n RIEPILOGO DI TUTTE LE PAGINE");
                printf(" DELL'INDICE E DELL'ARCHIVIO\n");
```

```
        getch();
        int n = 0;
        while (n < informazioni.indice.pagina_vuota_infondo - 1)
        {
                clrscr();
                stampa_nodo_file(n + 1);getch();
                ++n;
        }
        n = 0;
        while (n < informazioni.archivio.pagina_nonpiena_ultima)
        {
                clrscr();
                stampa_pagina_file(n + 1);getch();
                ++n;
        }

        }

        clrscr();
        printf("\n\n Ho inserito la %da chiave\n", i + 1);
        getch();
}

break;

case '2': // inserimento di una chiave

clrscr();
printf("\n\n\n INSERIMENTO DI UNA CHIAVE\n");

Tchiave chiave;
char co[20];
char no[20];

printf("\n Immetti la chiave da inserire: ");
scanf("%ld", &chiave);
printf(" Immetti il cognome da inserire: ");
scanf("%s", co);
printf(" Immetti il nome da inserire: ");
scanf("%s", no);

inserisci_chiave(chiave, informazioni.indice.pagina_radice, co, no);

stampa_albero();

clrscr();
printf("\n Vuoi vedere un riepilogo");
printf(" di tutte le pagine (s/n) ? ");

char w;
do
{
        w = getch();
        if (w != 's' && w != 'n')
                printf("\a");
} while (w != 's' && w != 'n');

if (w == 's')
{

clrscr();
printf("\n RIEPILOGO DI TUTTE LE PAGINE");
printf(" DELL'INDICE E DELL'ARCHIVIO\n");
getch();
int n = 0;
while (n < informazioni.indice.pagina_vuota_infondo - 1)
{
        clrscr();
        stampa_nodo_file(n + 1);getch();
        ++n;
}
n = 0;
while (n < informazioni.archivio.pagina_nonpiena_ultima)
{
```

```
        clrscr();
        stampa_pagina_file(n + 1);getch();
        ++n;
}

}

break;

case '3': // cancellazione di una chiave

clrscr();
printf("\n\n\n CANCELLAZIONE DI UNA CHIAVE\n");

printf("\n Immetti la chiave da cancellare: ");
scanf("%ld", &chiave);

cancella_chiave(chiave, informazioni.indice.pagina_radice);

stampa_albero();

clrscr();
printf("\n Vuoi vedere un riepilogo");
printf(" di tutte le pagine (s/n) ? ");

do
{

        w = getch();
        if (w != 's' && w != 'n')
                printf("\a");
} while (w != 's' && w != 'n');

if (w == 's')
{

clrscr();
printf("\n RIEPILOGO DI TUTTE LE PAGINE");
printf(" DELL'INDICE E DELL'ARCHIVIO\n");
getch();
int n = 0;
while (n < informazioni.indice.pagina_vuota_infondo - 1)
{
        clrscr();
        stampa_nodo_file(n + 1);getch();
        ++n;
}
n = 0;
while (n < informazioni.archivio.pagina_nonpiena_ultima)
{
        clrscr();
        stampa_pagina_file(n + 1);getch();
        ++n;
}

}

break;

case '4': // modifica di una chiave

clrscr();
printf("\n\n\n MODIFICA DI UNA CHIAVE\n");

printf("\n Immetti la chiave da modificare: ");
scanf("%ld", &chiave);
printf(" Immetti il nuovo cognome: ");
scanf("%s", co);
printf(" Immetti il nuovo nome: ");
scanf("%s", no);

modifica_dati(chiave, informazioni.indice.pagina_radice, co, no);

clrscr();
```

```
        printf("\n Vuoi vedere un riepilogo");
        printf(" di tutte le pagine dell'archivio (s/n) ? ");

        do
        {
               w = getch();
               if (w != 's' && w != 'n')
                      printf("\a");
        } while (w != 's' && w != 'n');

        if (w == 's')
        {

        int n = 0;
        while (n < informazioni.archivio.pagina_nonpiena_ultima)
        {
               clrscr();
               stampa_pagina_file(n + 1);getch();
               ++n;
        }

        }

        break;

        case '5': // ricerca di una chiave

        clrscr();
        printf("\n\n\n RICERCA DI UNA CHIAVE\n");

        printf("\n Immetti la chiave da ricercare: ");
        scanf("%ld", &chiave);

        Tnumpagina np;
        np = ricerca_chiave(chiave, informazioni.indice.pagina_radice);

        if (np == -1L)
        {
               printf(": chiave assente.\n");
               getch();
        }
        else if (np == 0L)
        {
               printf("\n La chiave non è presente\n");
               getch();
        }
        else
        {
               Tpuntpagina ppag = carica_pagina(np);
               int n = 0;
               while ((*ppag)[n].chiave != chiave)
                      ++n;
               printf("\n Il cognome corrispondente è: %s.",
                                                    (*ppag)[n].cognome);
               printf("\n Il nome corrispondente è: %s.\n", (*ppag)[n].nome);
               free(ppag);
               getch();
        }

        break;

        default:

        break;
}

clrscr();
printf("\n RIEPILOGO INFORMAZIONI SUI FILES DELL'INDICE E DELL'ARCHIVIO");
stampa_informazioni(informazioni);
getch();
```

```
        salva_informazioni(informazioni);

        clrscr();
        printf("\n\n Vuoi effettuare altre operazioni (s/n) ? ");

        do
        {
                r = getch();
                if (r != 's' && r != 'n')
                        printf("\a");
        } while (r != 's' && r != 'n');

        if (r == 's')
                printf("s");
        else
                printf("n");

        } while (r == 's');

        return 0;
}

/*************************** FINE PROGRAMMA PRINCIPALE ***************************/

/********************************* PRESENTAZIONE *********************************/

void presentazione(void)
{
        clrscr();
        gotoxy(15, 5);
        printf("Elaborato per l'esame di Documentazione Automatica");
        gotoxy(34, 11);
        printf("B+ Alberi");
        gotoxy(30, 13);
        printf("di Menchetti Sauro");
        gotoxy(33, 22);
        printf("A.A. 1997/98");
        gotoxy(80, 25);
        getch();
}

/************************ GENERAZIONE DI UNA CHIAVE CASUALE ************************/

Tchiave chiave_casuale(void)
{
        time_t t;

        srand((unsigned) time(&t));
        return (rand() % MAX_CASUALE);
}

/**************** FUNZIONI CHE GESTISCONO LE INFORMAZIONI SUI FILES ****************/

void stampa_informazioni(Tinfo_file_indice_archivio inf)
{
        printf("\n STAMPA DELLE INFORMAZIONI SUI FILES");
        printf(" DELL'INDICE E DELL'ARCHIVIO\n");
        printf("\n Informazioni FILE_INDICE");
        printf("\n Pagina della radice: %ld", inf.indice.pagina_radice);
        printf("\n Prima pagina sequence set: %ld", inf.indice.pagina_sequence_set);
        printf("\n Ultima pagina vuota: %ld", inf.indice.pagina_vuota_infondo);
        printf("\n Pagine cancellate: ");
        for (int i = 0; i < MAXDEL; ++i)
                if (inf.indice.pagine_cancellate[i] != 0L)
                        printf("%ld ", inf.indice.pagine_cancellate[i]);
        printf("\n\n Informazioni FILE_ARCHIVIO");
        printf("\n Ultima pagina non piena:");
        printf(" %ld", inf.archivio.pagina_nonpiena_ultima);
        printf("\n Pagine con cancellazioni: ");
        for (i = 0; i < MAXDEL; ++i)
                if (inf.archivio.pagine_cancellazioni[i] != 0L)
                        printf("%ld ", inf.archivio.pagine_cancellazioni[i]);
        printf("\n");
}
```

```
void salva_informazioni(Tinfo_file_indice_archivio inf)
{
	FILE *fp;

	if ((fp = fopen(FILE_INFORMAZIONI, "wb")) == NULL)
	{
		fprintf(stderr, "\n Non posso scrivere su %s", FILE_INFORMAZIONI);
		exit(1);
	}
	else
	{
		fwrite(&inf, sizeof(Tinfo_file_indice_archivio), 1, fp);
		if (ferror(fp))
		{
			fprintf(stderr, "\n Errore di scrittura");
			fprintf(stderr, " su %s", FILE_INFORMAZIONI);
			exit(2);
		}
		fclose(fp);
	}
}

void carica_informazioni(Tinfo_file_indice_archivio *pinf)
{
	FILE *fp;

	clrscr();
	fprintf(stderr, "\n Caricamento delle informazioni sui files…");
	if ((fp = fopen(FILE_INFORMAZIONI, "rb")) == NULL)
	{
		fprintf(stderr, "\n Archivio vuoto.");
		fprintf(stderr, "\n Il file %s", FILE_INFORMAZIONI);
		fprintf(stderr, " non è stato ancora creato:");
		fprintf(stderr, " non è disponibile alcuna\n informazione");
		fprintf(stderr, " sui files dell'indice e dell'archivio.");
	}
	else
	{
		fprintf(stderr, "fatto.");
		fread(pinf, sizeof(Tinfo_file_indice_archivio), 1, fp);
		if (ferror(fp))
		{
			fprintf(stderr, "\n Errore in lettura");
			fprintf(stderr, " su %s", FILE_INFORMAZIONI);
			exit(3);
		}
		fclose(fp);
	}
	gotoxy(80, 25);
	getch();
}

/******************** FUNZIONE CHE STAMPA UN NODO DELL'INDICE ********************/

void stampa_nodo(Tpuntnodo punt)
{
	printf("\n STAMPA DELLE INFORMAZIONI DI UN NODO\n");
	printf("\n Numero della pagina in cui è memorizzato: %ld", punt->np);
	printf("\n Nodo %s", punt->int_ter ? "terminale" : "intermedio");
	if (punt->int_ter) // se è terminale
	{
		printf("\n\n Numero della pagina precedente: %ld", punt->npp);
		printf("\n Numero della pagina successiva: %ld", punt->nps);
		for (int i = 0; i < ORDINE_M; ++i)
		{
			printf("\n\n Chiave:");
			if (punt->info[i].chiave == INFINITO)
				printf(" ∞");
			else
				printf(" %ld", punt->info[i].chiave);
			printf("\n Numero della pagina con chiave %ld: %ld",
				punt->info[i].chiave, punt->info[i].nptr);
		}
```

```
        }
        else // se è intermedio
        {
                for (int i = 0; i < ORDINE_M; ++i)
                {
                        printf("\n\n Chiave:");
                        if (punt->info[i].chiave == INFINITO)
                                printf(" ∞");
                        else
                                printf(" %ld", punt->info[i].chiave);
                        printf("\n Numero della pagina");
                        printf(" del figlio con chiavi <=");
                        if (punt->info[i].chiave == INFINITO)
                                printf(" ∞:");
                        else
                                printf(" %ld:", punt->info[i].chiave);
                        printf(" %ld", punt->info[i].nptr);
                }
        }
        printf("\n");
}

/***************** FUNZIONI CHE GESTISCONO I NODI DEL FILE_INDICE *****************/

void salva_nodo(Tpuntnodo punt, Tnumpagina num)
{
        FILE *fp;

        if (num <= 0)
        {
                printf("\n Errore in salva_nodo: pagina <= 0");
                exit(4);
        }
        if ((fp = fopen(FILE_INDICE, "r+b")) == NULL)
        {
                fprintf(stderr, "\n Non posso scrivere su %s", FILE_INDICE);
                exit(5);
        }
        else
        {
                if (fseek(fp, sizeof(Tnodo) * (num - 1), SEEK_SET))
                {
                        fprintf(stderr, "\n Errore di posizione su %s", FILE_INDICE);
                        exit(6);
                }
                fwrite(punt, sizeof(Tnodo), 1, fp);
                if (ferror(fp))
                {
                        fprintf(stderr, "\n Errore di scrittura su %s", FILE_INDICE);
                        exit(7);
                }
                fclose(fp);
        }
        free(punt);
}
```

```
Tpuntnodo carica_nodo(Tnumpagina num)
{
        FILE *fp;
        Tpuntnodo pnodo;

        if (num <= 0)
        {
                printf("\n Errore in carica_nodo: pagina <= 0");
                exit(8);
        }
        if ((pnodo = (Tpuntnodo)malloc(sizeof(Tnodo))) == NULL)
        {
           printf("\n Memoria non sufficiente per allocare il puntatore");
           exit(9);
        }
        if ((fp = fopen(FILE_INDICE, "rb")) == NULL)
        {
                fprintf(stderr, "\n Non posso leggere %s", FILE_INDICE);
                exit(10);
        }
        else
        {
                if (fseek(fp, sizeof(Tnodo) * (num - 1), SEEK_SET))
                {
                        fprintf(stderr, "\n Errore di posizione su %s", FILE_INDICE);
                        exit(11);
                }
                fread(pnodo, sizeof(Tnodo), 1, fp);
                if (ferror(fp))
                {
                        fprintf(stderr, "\n Errore in lettura su %s", FILE_INDICE);
                        exit(12);
                }
                fclose(fp);
        }
        return pnodo;
}

void stampa_nodo_file(Tnumpagina num)
{
        Tpuntnodo punt = carica_nodo(num);
        stampa_nodo(punt);
        free(punt);
}
```

```
/******************** DATI E FUNZIONI CHE GESTISCONO LO STACK ********************/

#define MAXVAL 20                   // massima profondità dello stack

int sp;                             // prossima posizione libera: stack pointer
Tnumpagina stack[MAXVAL];           // stack del numero delle pagine:
                                    // variabile esterna inizializzata a zero

void push(Tnumpagina pag) // push: inserisce pag in cima allo stack
{
        if (sp < MAXVAL)
                stack[sp++] = pag;
        else
        {
                printf("\n Errore: lo stack è pieno");
                printf("\n Aumentare MAXVAL");
                getch();
        }
}

Tnumpagina pop(void) // pop: preleva e ritorna il valore in cima allo stack
{
        if (sp > 0)
                return stack[--sp];
        else
        {
                printf("\n Errore: lo stack è vuoto");
                getch();
                return 0L;
        }
}

/********************** RICERCA BINARIA IN UN NODO TERMINALE **********************/

typedef int Indice_nodo;      // indice all'interno di un nodo: va da 0 a ORDINE_M - 1

Indice_nodo ricerca_binaria(Tchiave chiave, Tchiave v[], int num_chiavi)
{
        Indice_nodo low, high, mid;

        low = 0;
        high = num_chiavi - 1;
        while (low <= high)
        {
                mid = (low + high ) / 2;
                if (chiave < v[mid])
                        high = mid - 1;
                else if (chiave > v[mid])
                        low = mid + 1;
                else
                        return mid;
        }
        return -1;
}
```

```
/****************************** RICERCA DI UNA CHIAVE ******************************/
/*
restituisce:
        -1L se l'archivio è vuoto
        0L se la chiave non è presente
        il numero della pagina dell'archivio in cui si trova la chiave
        salva anche i nodi  che costituiscono il cammino
        dalla radice alla foglia nello stack
*/

Tnumpagina ricerca_chiave(Tchiave chiave, Tnumpagina pag_radice)
{
        Tpuntnodo pp;
        Tpuntnodo temp;
        Indice_nodo indice;

        sp = 0;
        if (pag_radice == 0L)
        {
                printf("\n L'archivio è ancora vuoto");
                return -1L;
        }
        else
        {
                pp = carica_nodo(pag_radice); // pp punta alla radice
                push(pp->np);
                while (!pp->int_ter) // finchè non siamo in un nodo terminale
                {
                        int i = 0;
                        while (chiave > pp->info[i].chiave)
                                ++i;
                        temp = carica_nodo(pp->info[i].nptr);
                        free(pp);
                        pp = temp;
                        push(pp->np);
                }
                // adesso siamo in un nodo terminale
                int n = 0; // indica il numero di chiavi realmente presenti nel nodo

                Tchiave vet[ORDINE_M];  // è un buffer temporaneo per la ricerca
                for (int i = 0; i < ORDINE_M; ++i)
                {
                        vet[i] = pp->info[i].chiave;
                        if (vet[i] != -1L)
                                ++n;
                }
                if ((indice = ricerca_binaria(chiave, vet, n)) != -1)
                {
                        Tnumpagina pag = pp->info[indice].nptr;
                        free(pp);
                        return pag;        // chiave trovata

                }
                else
                {
                        free(pp);
                        return 0L;         // chiave non trovata
                }
        }
}

/****************** FUNZIONE CHE STAMPA UNA PAGINA DELL'ARCHIVIO ******************/

void stampa_pagina(Tpagina pag)
{
        printf("\n STAMPA DEI DATI DI UNA PAGINA\n");
        for (int i = 0; i < NUM_DATI; ++i)
        {
                printf("\n %2d) Chiave  :", i + 1);
                if (pag[i].chiave == INFINITO)
                        printf(" ∞");
                else
                        printf(" %ld", pag[i].chiave);
```

```
                printf("\n     Cognome : %s", pag[i].cognome);
                printf("\n     Nome    : %s", pag[i].nome);
        }
        printf("\n");
}

/***************** FUNZIONI CHE GESTISCONO LE PAGINE DELL'ARCHIVIO *****************/

void salva_pagina(Tpuntpagina pg, Tnumpagina num)
{
        FILE *fp;

        if (num <= 0)
        {
                printf("\n Errore in salva_nodo: pagina <= 0");
                exit(13);
        }
        if ((fp = fopen(FILE_ARCHIVIO, "r+b")) == NULL)
        {
                fprintf(stderr, "\n Non posso scrivere su %s", FILE_ARCHIVIO);
                exit(14);
        }
        else
        {
                if (fseek(fp, sizeof(Tpagina) * (num - 1), SEEK_SET))
                {
                        fprintf(stderr, "\n Errore di posizione");
                        fprintf(stderr, " su %s", FILE_ARCHIVIO);
                        exit(15);
                }
                fwrite(pg, sizeof(Tpagina), 1, fp);
                if (ferror(fp))
                {
                        fprintf(stderr, "\n Errore di scrittura");
                        fprintf(stderr, " su %s", FILE_ARCHIVIO);
                        exit(16);
                }
                fclose(fp);
        }
        free(pg);
}

Tpuntpagina carica_pagina(Tnumpagina num)
{
        FILE *fp;
        Tpuntpagina ppag;

        if (num <= 0)
        {
                printf("\n Errore in carica_pagina: pagina <= 0");
                exit(17);
        }
        if ((ppag = (Tpuntpagina)malloc(sizeof(Tpagina))) == NULL)
        {
           printf("\n Memoria non sufficiente per allocare il puntatore");
           exit(18);
        }
        if ((fp = fopen(FILE_ARCHIVIO, "rb")) == NULL)
        {
                fprintf(stderr, "\n Non posso leggere %s", FILE_ARCHIVIO);
                exit(19);
        }
        else
        {
                if (fseek(fp, sizeof(Tpagina) * (num - 1), SEEK_SET))
                {
                        fprintf(stderr, "\n Errore di posizione");
                        fprintf(stderr, " su %s", FILE_ARCHIVIO);
                        exit(20);
                }
                fread(ppag, sizeof(Tpagina), 1, fp);
                if (ferror(fp))
                {
```

```
                        fprintf(stderr, "\n Errore in lettura su %s", FILE_ARCHIVIO);
                        exit(21);
                }
                fclose(fp);
        }
        return ppag;
}

void stampa_pagina_file(Tnumpagina num)
{
        Tpuntpagina ppag = carica_pagina(num);
        stampa_pagina(*ppag);
        free(ppag);
}

/******************** CREA I FILES FILE_ARCHIVIO E FILE_INDICE ********************/

void crea_files(void)
{
        FILE *fp;

        if ((fp = fopen(FILE_INDICE, "rb")) == NULL)
        {
                fp = fopen(FILE_INDICE, "wb");
                fclose(fp);
        }
        if ((fp = fopen(FILE_ARCHIVIO, "rb")) == NULL)
        {
                fp = fopen(FILE_ARCHIVIO, "wb");
                fclose(fp);
        }
}

/******************************** INIZIALIZZAZIONI ********************************/

// valore delle chiavi = -1L; numero delle pagine =  0L

void inizializza_nodo(Tpuntnodo p)
{
        p->np  = 0L;
        p->int_ter = NODO_TERMINALE;
        p->npp = 0L;
        p->nps = 0L;
        for (int i = 0; i < ORDINE_M; ++i)
        {
                p->info[i].chiave = -1L;
                p->info[i].nptr   =  0L;
        }
}

void inizializza_pagina(Tpuntpagina p)
{
        for (int i = 0; i < NUM_DATI; ++i)
        {
                (*p)[i].chiave  = -1L;
                strcpy((*p)[i].cognome, "");
                strcpy((*p)[i].nome,    "");
        }
}

/********************** TEST PER VEDERE SE LA PAGINA E' PIENA **********************/

Boolean check_pagina_piena(Tnumpagina npag) // 1 piena, 0 non piena
{
        Tpuntpagina ppg = carica_pagina(npag);
        for (int i = 0; i < NUM_DATI; ++i)
                if ((*ppg)[i].chiave == -1L)
                {
                        free(ppg);
                        return 0;
                }
        free(ppg);
        return 1;
}
```

```
/*************** FUNZIONI CHE RESTITUISCONO LA PRIMA PAGINA LIBERA ***************/

Tnumpagina prima_pagina_archivio(void)
{
       Tnumpagina npg = informazioni.archivio.pagina_nonpiena_ultima;
       for (int i = 0; i < MAXDEL; ++i)
              if ((informazioni.archivio.pagine_cancellazioni[i] != 0L)
                   && (informazioni.archivio.pagine_cancellazioni[i] < npg))
                     npg = informazioni.archivio.pagine_cancellazioni[i];
       return  npg;
}

void verifica_pagina_piena(Tnumpagina nppa)
{
       if (check_pagina_piena(nppa))
       {
              if (nppa == informazioni.archivio.pagina_nonpiena_ultima)
              {
                     informazioni.archivio.pagina_nonpiena_ultima += 1;
                     Tpuntpagina pagina;
                     if ((pagina = (Tpuntpagina)malloc(sizeof(Tpagina))) == NULL)
                     {
                            printf("\n Memoria non sufficiente");
                            printf(" per allocare il puntatore");
                            exit(22);
                     }
                     inizializza_pagina(pagina);
                     salva_pagina(pagina,
                                  informazioni.archivio.pagina_nonpiena_ultima);
              }
              for (int i = 0; i < MAXDEL; ++i)
                     if (nppa == informazioni.archivio.pagine_cancellazioni[i])
                            informazioni.archivio.pagine_cancellazioni[i] = 0L;
       }
}

Tnumpagina prima_pagina_indice(void)
{
       Tnumpagina npg = informazioni.indice.pagina_vuota_infondo;
       for (int i = 0; i < MAXDEL; ++i)
              if ((informazioni.indice.pagine_cancellate[i] != 0L)
                   && (informazioni.indice.pagine_cancellate[i] < npg))
                     npg = informazioni.indice.pagine_cancellate[i];

       if (npg == informazioni.indice.pagina_vuota_infondo)
       {
              informazioni.indice.pagina_vuota_infondo += 1;
              Tpuntnodo p;
              if ((p = (Tpuntnodo)malloc(sizeof(Tnodo))) == NULL)
              {
                 printf("\n Memoria non sufficiente per allocare il puntatore");
                 exit(23);
              }
              inizializza_nodo(p);
              salva_nodo(p, informazioni.indice.pagina_vuota_infondo);
       }
       for (i = 0; i < MAXDEL; ++i)
              if (npg == informazioni.indice.pagine_cancellate[i])
                     informazioni.indice.pagine_cancellate[i] = 0L;
       return  npg;
}

/************** DICHIARAZIONE E GESTIONE DEI BUFFER PER L'INSERIMENTO **************/

typedef struct buffer {                  // buffer per la radice piena
       Tchiave chiave;
       Tnumpagina nptr;
       } Vet[ORDINE_M + 1];

typedef struct buffer_rid {              // buffer per la ridistribuzione
       Tchiave chiave;
       Tnumpagina nptr;
       } Vet_rid[2 * ORDINE_M];
```

```
typedef struct buffer_adiac {            // buffer per lo split
        Tchiave chiave;
        Tnumpagina nptr;
        } Vet_adiac[2 * ORDINE_M + 1];

// i buffer non devono essere vuoti quando ci inserisco

void inserisci_ord_nel_buffer(Tchiave chiave, Vet v, Tnumpagina n, int dimbuf)
{
        int i = 0;
        while ((chiave > v[i].chiave) && (v[i].chiave != -1L))
                ++i;
        int posizione = i;
        for (i = dimbuf - 2; i >= posizione; --i)
        {
                v[i + 1].chiave = v[i].chiave;
                v[i + 1].nptr   = v[i].nptr;
        }
        v[posizione].chiave = chiave;
        v[posizione].nptr   = n;
}

void inserisci_ord_nel_buffer_rid(Tchiave ch, Vet_rid v, Tnumpagina n, int dimbuf)
{
        int i = 0;
        while ((ch > v[i].chiave) && (v[i].chiave != -1L))
                ++i;
        int posizione = i;
        for (i = dimbuf - 2; i >= posizione; --i)
        {
                v[i + 1].chiave = v[i].chiave;
                v[i + 1].nptr   = v[i].nptr;
        }
        v[posizione].chiave = ch;
        v[posizione].nptr   = n;
}

void inserisci_ord_nel_buffer_adiac(Tchiave c, Vet_adiac v, Tnumpagina n, int dimbuf)
{
        int i = 0;
        while ((c > v[i].chiave) && (v[i].chiave != -1L))
                ++i;
        int posizione = i;
        for (i = dimbuf - 2; i >= posizione; --i)
        {
                v[i + 1].chiave = v[i].chiave;
                v[i + 1].nptr   = v[i].nptr;
        }
        v[posizione].chiave = c;
        v[posizione].nptr   = n;
}

/************* FUNZIONE CHE INSERISCE ORDINATAMENTE LA CHIAVE NEL NODO *************/

// il nodo non deve essere vuoto quando ci inserisco la chiave

void inserisci_ord_nel_nodo(Tchiave chiave, Tpuntnodo pn, Tnumpagina n)
{
        int i = 0;
        while ((chiave > pn->info[i].chiave) && (pn->info[i].chiave != -1L))
                ++i;
        int posizione = i;
        for (i = ORDINE_M - 2; i >= posizione; --i)
        {
                pn->info[i + 1].chiave = pn->info[i].chiave;
                pn->info[i + 1].nptr   = pn->info[i].nptr;
        }
        pn->info[posizione].chiave = chiave;
        pn->info[posizione].nptr   = n;
}
```

```
/*************** INSERIMENTO DEL DATO NELLA PAGINA DI FILE_ARCHIVIO ***************/

void inserisci_nella_pagina(Tchiave chiave, char cog[20], char nome[20], Tpagina pag)
{
	int i = 0;
	while (pag[i].chiave != -1L)		// inserisce nella prima
		++i;				// posizione libera
	pag[i].chiave  =  chiave;
	strcpy(pag[i].cognome, cog);
	strcpy(pag[i].nome, nome);
}

/**************** CERCA NEL PADRE L'INDICE DEL PUNTATORE AL FIGLIO *****************/

Indice_nodo cerca(Tpuntnodo padre, Tnumpagina np)
{
	for (int i = 0; i < ORDINE_M; ++i)
		if (padre->info[i].nptr == np)
				return i;
	return ORDINE_M; // condizione di errore
}

/*************** FUNZIONE CHE CERCA UN FRATELLO ADIACENTE NON PIENO ***************/

// pp punta al nodo pieno in cui inserire la chiave che non può essere la radice
// restituisce il puntatore al fratello non pieno oppure NULL se non esiste
// padre è il nodo padre di pp: c'è sempre in quanto pp non può essere la radice

Tpuntnodo ridistribuzione_ins(Tpuntnodo pp)
{
	Tpuntnodo padre = carica_nodo(pop());
	push(padre->np);
	Indice_nodo indice = cerca(padre, pp->np);
	if (indice == ORDINE_M)
	{
		printf("\n Errore: ridistribuzione inserimento");
		getch();
		return NULL;
	}
	else if (indice == 0)
	{
		Tpuntnodo pfra = carica_nodo(padre->info[++indice].nptr);
		if (pfra->info[ORDINE_M - 1].chiave == -1L)
		{
			free(padre);
			return pfra;
		}
		else
		{
			free(padre);
			free(pfra);
			return NULL;
		}
	}
	else if (indice == ORDINE_M - 1)
	{
		Tpuntnodo pfra = carica_nodo(padre->info[--indice].nptr);
		if (pfra->info[ORDINE_M - 1].chiave == -1L)
		{
			free(padre);
			return pfra;
		}
		else
		{
			free(padre);
			free(pfra);
			return NULL;
		}
	}
	else // ci sono due fratelli
	{
		Indice_nodo indice1 = indice + 1;
		Indice_nodo indice2 = indice - 1;
		Tpuntnodo pfra1 = NULL;
```

```
Tpuntnodo pfra2 = NULL;
if (padre->info[indice1].chiave >= 0)
       pfra1 = carica_nodo(padre->info[indice1].nptr);
if (padre->info[indice2].chiave >= 0)
       pfra2 = carica_nodo(padre->info[indice2].nptr);

if ((pfra1 != NULL) && (pfra2 != NULL))
{
       if ((pfra1->info[ORDINE_M - 1].chiave == -1L)
            && (pfra2->info[ORDINE_M - 1].chiave == -1L))
       {
              time_t t;
              srand((unsigned) time(&t));
              Boolean ret = rand() % 2;
              if (ret)
              {
                     free(padre);
                     free(pfra2);
                     return pfra1;
              }
              else
              {
                     free(padre);
                     free(pfra1);
                     return pfra2;
              }
       }
       else if (pfra1->info[ORDINE_M - 1].chiave == -1L)
       {
              free(padre);
              free(pfra2);
              return pfra1;
       }
       else if (pfra2->info[ORDINE_M - 1].chiave == -1L)
       {
              free(padre);
              free(pfra1);
              return pfra2;
       }
       else
       {
              free(padre);
              free(pfra1);
              free(pfra2);
              return NULL;
       }
}
else if (pfra1 != NULL)
{
       if (pfra1->info[ORDINE_M - 1].chiave == -1L)
       {
              free(padre);
              return pfra1;
       }
       else
       {
              free(padre);
              free(pfra1);
              return NULL;
       }
}
else if (pfra2 != NULL)
{
       if (pfra2->info[ORDINE_M - 1].chiave == -1L)
       {
              free(padre);
              return pfra2;
       }
       else
       {
              free(padre);
              free(pfra2);
              return NULL;
       }
```

```
                }
                else
                {
                        printf("\n Errore: ridistribuzione inserimento");
                        getch();
                        return NULL;
                }
        }
}

/***************** FUNZIONE CHE RESTITUISCE UN FRATELLO ADIACENTE ******************/

Tpuntnodo fratello_adiacente(Tpuntnodo pp)
{
        Tpuntnodo padre = carica_nodo(pop());
        push(padre->np);
        Indice_nodo indice = cerca(padre, pp->np);
        if (indice == ORDINE_M)
        {
                printf("\n Errore: fratello adiacente pieno");
                getch();
                return NULL;
        }
        else if (indice == 0)
        {
                Tpuntnodo pfra = carica_nodo(padre->info[++indice].nptr);
                {
                        free(padre);
                        return pfra;
                }
        }
        else if (indice == ORDINE_M - 1)
        {
                Tpuntnodo pfra = carica_nodo(padre->info[--indice].nptr);
                {
                        free(padre);
                        return pfra;
                }
        }
        else // ci sono due fratelli
        {
                Indice_nodo indice1 = indice + 1;
                Indice_nodo indice2 = indice - 1;
                Tpuntnodo pfra1 = NULL;
                Tpuntnodo pfra2 = NULL;
                if (padre->info[indice1].chiave >= 0)
                        pfra1 = carica_nodo(padre->info[indice1].nptr);
                if (padre->info[indice2].chiave >= 0)
                        pfra2 = carica_nodo(padre->info[indice2].nptr);

                if ((pfra1 != NULL) && (pfra2 != NULL))
                {
                        time_t t;
                        srand((unsigned) time(&t));
                        Boolean ret = rand() % 2;
                        if (ret)
                        {
                                free(padre);
                                free(pfra2);
                                return pfra1;
                        }
                        else
                        {
                                free(padre);
                                free(pfra1);
                                return pfra2;
                        }
                }
                else if (pfra1 != NULL)
                {
                        free(padre);
                        return pfra1;
                }
                else if (pfra2 != NULL)
```

```
                {
                        free(padre);
                        return pfra2;
                }
                else
                {
                        printf("\n Errore: fratello adiacente pieno");
                        getch();
                        return NULL;
                }
        }
}

/****************** PROCEDURE VARIE DI INSERIMENTO DI UNA CHIAVE ******************/

void inserisci_nodo_non_pieno(Tchiave c, char cog[20], char nom[20],
                                                Tpuntnodo puntnodo, Tnumpagina nnp)
{
        Tnumpagina nppla;
        Tpuntpagina ppag;
        if (puntnodo->int_ter)
        {
                nppla = prima_pagina_archivio();
                ppag = carica_pagina(nppla);
        }

        inserisci_ord_nel_nodo(c, puntnodo, (puntnodo->int_ter) ? nppla : nnp);

        if (puntnodo->int_ter)
        {
                inserisci_nella_pagina(c, cog, nom, *ppag);
                salva_pagina(ppag, nppla);
                verifica_pagina_piena(nppla);
        }
        salva_nodo(puntnodo, puntnodo->np);
}

void inserisci_radice_piena(Tchiave c, char cog[20], char nom[20],
                                                Tpuntnodo pt, Tnumpagina nnp)
{
        Vet vet;
        for (int i = 0; i < ORDINE_M; ++i)
        {
                vet[i].chiave = pt->info[i].chiave;
                vet[i].nptr = pt->info[i].nptr;
        }
        vet[ORDINE_M].chiave = -1L;
        vet[ORDINE_M].nptr = 0L;

        Tnumpagina nppla;
        Tpuntpagina ppag;
        if (pt->int_ter)
        {
                nppla = prima_pagina_archivio();
                ppag = carica_pagina(nppla);
        }

        inserisci_ord_nel_buffer(c, vet, (pt->int_ter) ? nppla : nnp, ORDINE_M + 1);

        for (i = 0; i < ORDINE_M / 2 + 1; ++i)
        {
                pt->info[i].chiave = vet[i].chiave;
                pt->info[i].nptr   = vet[i].nptr;
        }
        for (i = ORDINE_M / 2 + 1; i < ORDINE_M ; ++i)
        {
                pt->info[i].chiave = -1L;
                pt->info[i].nptr   = 0L;
        }

        Tnumpagina npp1 = prima_pagina_indice();
        Tnumpagina npp2 = prima_pagina_indice();
        Tpuntnodo pp1 = carica_nodo(npp1);   // è quello terminale
        Tpuntnodo pp2 = carica_nodo(npp2);   // è la nuova radice
```

```
        pp1->np = npp1;
        if (pt->int_ter)
                pp1->npp = pt->np;
        else
                pp1->int_ter = NODO_INTERMEDIO;
        int j;
        for (j = 0, i = ORDINE_M / 2 + 1; i < ORDINE_M + 1; ++j, ++i)
        {
                pp1->info[j].chiave = vet[i].chiave;
                pp1->info[j].nptr   = vet[i].nptr;
        }
        pp2->np = npp2;
        pp2->int_ter = NODO_INTERMEDIO;
        pp2->info[0].chiave = vet[ORDINE_M / 2].chiave;
        pp2->info[0].nptr = pt->np;
        pp2->info[1].chiave = INFINITO;
        pp2->info[1].nptr = pp1->np;

        pt->nps = pp1->np;

        informazioni.indice.pagina_radice = pp2->np;

        if (pt->int_ter)
        {
                inserisci_nella_pagina(c, cog, nom, *ppag);
                salva_pagina(ppag, nppla);
                verifica_pagina_piena(nppla);
        }
        salva_nodo(pt, pt->np);
        salva_nodo(pp1, pp1->np);
        salva_nodo(pp2, pp2->np);
}

void inserisci_ridistribuzione(Tchiave c, char cog[20], char nom[20],
                               Tpuntnodo puntnodo, Tnumpagina nnp, Tpuntnodo pfratello)
{
        Tpuntnodo ppadre = carica_nodo(pop());
        push(ppadre->np);

        Vet_rid vet;
        for (int i = 0; i < 2 * ORDINE_M; ++i)
        {
                vet[i].chiave = -1L;
                vet[i].nptr = 0L;
        }
        for (i = 0; i < ORDINE_M; ++i)
        {
                vet[i].chiave = puntnodo->info[i].chiave;
                vet[i].nptr = puntnodo->info[i].nptr;
        }
        for (i = 0; i < ORDINE_M; ++i)
                if ((pfratello->info[i].chiave) != -1L)
                        inserisci_ord_nel_buffer_rid(pfratello->info[i].chiave, vet,
                                                     pfratello->info[i].nptr, 2 * ORDINE_M);

        Tnumpagina nppla;
        Tpuntpagina ppag;
        if (puntnodo->int_ter)
        {
                nppla = prima_pagina_archivio();
                ppag = carica_pagina(nppla);
        }

        inserisci_ord_nel_buffer_rid(c, vet, (puntnodo->int_ter) ? nppla : nnp,
                                                                  2 * ORDINE_M);

        Boolean destra = (puntnodo->info[0].chiave < pfratello->info[0].chiave);

        if (destra)
        {
                for (i = 0; i < ORDINE_M; ++i)
                {
                        puntnodo->info[i].chiave = vet[i].chiave;
```

```
                        puntnodo->info[i].nptr = vet[i].nptr;
                }
                int j;
                for (j = 0, i = ORDINE_M; i < 2 * ORDINE_M; ++j, ++i)
                {
                        pfratello->info[j].chiave = vet[i].chiave;
                        pfratello->info[j].nptr = vet[i].nptr;
                }
                ppadre->info[cerca(ppadre, puntnodo->np)].chiave
                        = puntnodo->info[ORDINE_M - 1].chiave;
        }
        else
        {
                for (i = 0; i < ORDINE_M; ++i)
                {
                        pfratello->info[i].chiave = vet[i].chiave;
                        pfratello->info[i].nptr = vet[i].nptr;
                }
                int j;
                for (j = 0, i = ORDINE_M; i < 2 * ORDINE_M; ++j, ++i)
                {
                        puntnodo->info[j].chiave = vet[i].chiave;
                        puntnodo->info[j].nptr = vet[i].nptr;
                }
                ppadre->info[cerca(ppadre, pfratello->np)].chiave
                        = pfratello->info[ORDINE_M - 1].chiave;
        }

        if (puntnodo->int_ter)
        {
                inserisci_nella_pagina(c, cog, nom, *ppag);
                salva_pagina(ppag, nppla);
                verifica_pagina_piena(nppla);
        }
        salva_nodo(puntnodo, puntnodo->np);
        salva_nodo(ppadre, ppadre->np);
        salva_nodo(pfratello, pfratello->np);
}

void inserisci_split(Tchiave *c, char cog[20], char nom[20], Tpuntnodo *puntnodo,
                                                                Tnumpagina *nnp)
{
        Tpuntnodo pfratello = fratello_adiacente(*puntnodo);

        clrscr();
        printf(" Stampa del nodo con cui suddividere le chiavi\n");
        getch();
        stampa_nodo(pfratello);
        getch();

        Tpuntnodo ppadre = carica_nodo(pop());
        push(ppadre->np);

        Vet_adiac vet;
        for (int i = 0; i < 2 * ORDINE_M + 1; ++i)
        {
                vet[i].chiave = -1L;
                vet[i].nptr = 0L;
        }
        for (i = 0; i < ORDINE_M; ++i)
        {
                vet[i].chiave = (*puntnodo)->info[i].chiave;
                vet[i].nptr = (*puntnodo)->info[i].nptr;
        }
        for (i = 0; i < ORDINE_M; ++i)
                inserisci_ord_nel_buffer_adiac(pfratello->info[i].chiave, vet,
                                        pfratello->info[i].nptr, 2 * ORDINE_M + 1);

        Tnumpagina nppla;
        Tpuntpagina ppag;
        if ((*puntnodo)->int_ter)
        {
                nppla = prima_pagina_archivio();
                ppag = carica_pagina(nppla);
```

```
}

inserisci_ord_nel_buffer_adiac(*c, vet,
                  ((*puntnodo)->int_ter) ? nppla : *nnp, 2 * ORDINE_M + 1);

Tnumpagina npp1 = prima_pagina_indice();
Tpuntnodo pp1 = carica_nodo(npp1);

const int primo = (2 * ORDINE_M + 1) / 3;
const int secondo = primo + (2 * ORDINE_M + 1 + 1) / 3;

Boolean destra = ((*puntnodo)->info[0].chiave < pfratello->info[0].chiave);

for (i = 0; i < ORDINE_M; ++i)
{
       (*puntnodo)->info[i].chiave = -1L;
       (*puntnodo)->info[i].nptr   = 0L;
}
for (i = 0; i < ORDINE_M; ++i)
{
       pfratello->info[i].chiave = -1L;
       pfratello->info[i].nptr   = 0L;
}
if (destra)
{
       for (i = 0; i < primo; ++i)
       {
              (*puntnodo)->info[i].chiave = vet[i].chiave;
              (*puntnodo)->info[i].nptr   = vet[i].nptr;
       }
       int j;
       for (j = 0, i = secondo; i < 2 * ORDINE_M + 1; ++j, ++i)
       {
              pfratello->info[j].chiave = vet[i].chiave;
              pfratello->info[j].nptr   = vet[i].nptr;
       }
}
else
{
       for (i = 0; i < primo; ++i)
       {
              pfratello->info[i].chiave = vet[i].chiave;
              pfratello->info[i].nptr   = vet[i].nptr;
       }
       int j;
       for (j = 0, i = secondo; i < 2 * ORDINE_M + 1; ++j, ++i)
       {
              (*puntnodo)->info[j].chiave = vet[i].chiave;
              (*puntnodo)->info[j].nptr   = vet[i].nptr;
       }
}
int j;
for (j = 0, i = primo; i < secondo; ++j, ++i)
{
       pp1->info[j].chiave = vet[i].chiave;
       pp1->info[j].nptr   = vet[i].nptr;
}

pp1->np = npp1;

if ((*puntnodo)->int_ter)
{
       if (destra)
       {
              (*puntnodo)->nps = pp1->np;
              pp1->npp = (*puntnodo)->np;
              pp1->nps = pfratello->np;
              pfratello->npp = pp1->np;
       }
       else
       {
              (*puntnodo)->npp = pp1->np;
              pp1->npp = pfratello->np;
              pp1->nps = (*puntnodo)->np;
```

```
                        pfratello->nps = pp1->np;
                }
        }
        else
                pp1->int_ter = NODO_INTERMEDIO;

        if (destra)
                ppadre->info[cerca(ppadre, (*puntnodo)->np)].chiave
                        = (*puntnodo)->info[primo - 1].chiave;
        else
                ppadre->info[cerca(ppadre, pfratello->np)].chiave
                        = pfratello->info[primo - 1].chiave;

        if ((*puntnodo)->int_ter)
        {
                inserisci_nella_pagina(*c, cog, nom, *ppag);
                salva_pagina(ppag, nppla);
                verifica_pagina_piena(nppla);
        }

        (*c) = pp1->info[secondo - primo - 1].chiave;
        (*nnp) = pp1->np;

        salva_nodo((*puntnodo), (*puntnodo)->np);
        salva_nodo(pp1, pp1->np);
        salva_nodo(ppadre, ppadre->np);
        salva_nodo(pfratello, pfratello->np);

        (*puntnodo) = carica_nodo(pop());
}

/*************************** INSERIMENTO DI UNA CHIAVE ***************************/

/* inserimento di una chiave assente in un archivio già creato */

void inserimento(Tchiave chiave, char cognome[20], char nome[20])
{
        Tpuntnodo pp = carica_nodo(pop()); // pp punta al nodo in cui inserire
        Tnumpagina nnpag = 0; // serve in caso di propagazione ai livelli superiori
        Boolean termine = 0;

        while (!termine)
        {

        clrscr();
        printf(" Stampa del nodo in cui va inserita la chiave %ld\n", chiave);
        getch();
        stampa_nodo(pp);
        getch();

        if (pp->info[ORDINE_M - 1].chiave == -1L) /* nodo non pieno */
        {
                clrscr();
                printf("\n\n\n Inserimento in un nodo non pieno\n");
                getch();

                inserisci_nodo_non_pieno(chiave, cognome, nome, pp, nnpag);
                termine = 1;
        }
        else /* nodo pieno */
        {
                if (pp->np == informazioni.indice.pagina_radice) /* radice piena */
                {
                        clrscr();
                        printf("\n\n\n Inserimento nella radice piena\n");
                        getch();

                        inserisci_radice_piena(chiave, cognome, nome, pp, nnpag);
                        termine = 1;
                }
                else /* inserimento in un nodo pieno diverso dalla radice */
                {
                        Tpuntnodo pfratello = ridistribuzione_ins(pp);
                        if (pfratello != NULL) /* ridistribuzione */
```

```
                        {
                                clrscr();
                                printf("\n\n\n Inserimento con ridistribuzione\n");
                                getch();
                                clrscr();
                                printf(" Stampa del nodo");
                                printf(" con cui ridistribuire le chiavi\n");
                                getch();
                                stampa_nodo(pfratello);
                                getch();

                                inserisci_ridistribuzione(chiave, cognome, nome, pp,
                                                                        nnpag, pfratello);
                                termine = 1;
                        }
                        else /* split */
                        {
                                clrscr();
                                printf("\n\n\n Inserimento con split\n");
                                getch();

                                inserisci_split(&chiave, cognome, nome, &pp, &nnpag);
                        }
                }
        }

        } /* fine del while (!termine) */
}

/*************************** INSERIMENTO DI UNA CHIAVE ***************************/

void inserisci_chiave(Tchiave chiave, Tnumpagina radice, char cog[20], char nome[20])
{
        clrscr();
        printf("\n PROCEDURA DI INSERIMENTO DI UNA CHIAVE\n");
        printf("\n Inserimento della chiave %ld.", chiave);
        printf("\n Il cognome è %s, il nome è %s.\n", cog, nome);
        getch();

        Tnumpagina npag;

        if ((npag = ricerca_chiave(chiave, radice)) == -1L) /* archivio vuoto */
        {
                printf(" e quindi verrà creato.");
                printf("\n\n Creazione dell'archivio…");

                Tpuntpagina ppag;
                Tpuntnodo pnodo;

                if ((ppag = (Tpuntpagina)malloc(sizeof(Tpagina))) == NULL)
                {
                   printf("\n Memoria non sufficiente per allocare il puntatore");
                   exit(24);
                }
                if ((pnodo = (Tpuntnodo)malloc(sizeof(Tnodo))) == NULL)
                {
                   printf("\n Memoria non sufficiente per allocare il puntatore");
                   exit(25);
                }

                inizializza_nodo(pnodo);
                inizializza_pagina(ppag);

                crea_files();

                pnodo->np = 1L;
                pnodo->info[0].chiave = chiave;
                pnodo->info[0].nptr = 1L;
                pnodo->info[1].chiave = INFINITO;

                inserisci_nella_pagina(chiave, cog, nome, *ppag);

                informazioni.indice.pagina_radice       = 1L;
                informazioni.indice.pagina_sequence_set = 1L;
```

```
                informazioni.indice.pagina_vuota_infondo     = 2L;
                informazioni.archivio.pagina_nonpiena_ultima = 1L;

                salva_nodo(pnodo, 1L);
                salva_pagina(ppag, 1L);

                if ((pnodo = (Tpuntnodo)malloc(sizeof(Tnodo))) == NULL)
                {
                   printf("\n Memoria non sufficiente per allocare il puntatore");
                   exit(26);
                }
                inizializza_nodo(pnodo);
                salva_nodo(pnodo, 2L);

                printf("fatto.");
                getch();
        }
        else if (npag >= 1L) /* chiave presente */
        {
                printf("\n La chiave %ld è già presente", chiave);
                printf(" alla pagina %ld, quindi non verrà inserita.\n", npag);
                getch();
        }
        else
                inserimento(chiave, cog, nome);
}

/*************** FUNZIONE CHE CONTA IL NUMERO DI CHIAVI IN UN NODO ***************/

// anche il valore INFINITO è contato come chiave

int conta_chiavi_nodo(Tpuntnodo punt)
{
        int numero = 0;
        while ((numero < ORDINE_M) && (punt->info[numero].chiave != -1L))
                ++numero;

        if ((numero <=1) || (numero >= ORDINE_M + 1))
        {
                printf("\n Errore: numero di chiavi nel nodo sbagliato");
                getch();
        }

        return numero;
}

/****************** FUNZIONE CHE CANCELLA LA CHIAVE DA UN NODO ******************/

void cancella_dal_nodo(Tchiave chiave, Tpuntnodo pp)
{
        int i = 0;
        while (chiave > pp->info[i].chiave)
                ++i;
        int posizione = i;
        for (i = posizione; i < ORDINE_M - 1; ++i)
        {
                pp->info[i].chiave = pp->info[i + 1].chiave;
                pp->info[i].nptr   = pp->info[i + 1].nptr;
        }
        pp->info[ORDINE_M - 1].chiave = -1L;
        pp->info[ORDINE_M - 1].nptr   = 0L;
}

/****************** FUNZIONE CHE CANCELLA LA CHIAVE DA UNA PAGINA ******************/

void cancella_dalla_pagina(Tchiave chiave, Tpuntpagina puntpag)
{
        int i = 0;
        while ((*puntpag)[i].chiave != chiave)
                ++i;
        (*puntpag)[i].chiave = -1L;
        strcpy((*puntpag)[i].cognome, "");
        strcpy((*puntpag)[i].nome, "");
}
```

```
/******** FUNZIONE CHE GESTISCE LE CANCELLAZIONI NELLE PAGINE DELL'ARCHIVIO ********/

// viene usata ogni volta che faccio una cancellazione nelle pagine dell'archivio

void informazioni_archivio_cancellazioni(Tnumpagina pag)
{
	int i = 0;
	while ((i < MAXDEL) && (informazioni.archivio.pagine_cancellazioni[i] != 0L)
		&& (informazioni.archivio.pagine_cancellazioni[i] != pag))
		++i;
	if (i == MAXDEL)
	{
		printf("\n Troppe cancellazioni: aumentare MAXDEL");
		getch();
		goto fine;
	}
	informazioni.archivio.pagine_cancellazioni[i] = pag;
fine:
}

/********* FUNZIONE CHE GESTISCE LE CANCELLAZIONI DELLE PAGINE DELL'INDICE *********/

// viene usata solo quando faccio una concatenazione tra le pagine dell'indice

void informazioni_indice_cancellate(Tnumpagina pag)
{
	int i = 0;
	while ((i < MAXDEL) && (informazioni.indice.pagine_cancellate[i] != 0L)
		&& (informazioni.indice.pagine_cancellate[i] != pag))
		++i;
	if (i == MAXDEL)
	{
		printf("\n Troppe cancellazioni: aumentare MAXDEL");
		getch();
		goto fine;
	}
	informazioni.indice.pagine_cancellate[i] = pag;
fine:
}

/****** FUNZIONE CHE CERCA UN NODO ADIACENTE CON CUI RIDISTRIBUIRE LE CHIAVI *******/

// pp punta al nodo pieno in cui cancellare la chiave che non può essere la radice
// ritorna il puntatore al fratello con cui ridistribuire oppure NULL se non esiste
// padre è il nodo padre di pp: c'è sempre in quanto pp non può essere la radice

Tpuntnodo ridistribuzione_can(Tpuntnodo pp)
{
	Tpuntnodo padre = carica_nodo(pop());
	push(padre->np);

	Indice_nodo indice = cerca(padre, pp->np);
	if (indice == ORDINE_M)
	{
		printf("\n Errore: ridistribuzione cancellazione");
		getch();
		return NULL;
	}
	else if (indice == 0)
	{
		Tpuntnodo pfra = carica_nodo(padre->info[++indice].nptr);
		if (conta_chiavi_nodo(pfra) >= LIMITE_INF + 1)
		{
			free(padre);
			return pfra;
		}
		else
		{
			free(padre);
			free(pfra);
			return NULL;
		}
	}
```

```
else if (indice == ORDINE_M - 1)
{
	Tpuntnodo pfra = carica_nodo(padre->info[--indice].nptr);
	if (conta_chiavi_nodo(pfra) >= LIMITE_INF + 1)
	{
		free(padre);
		return pfra;
	}
	else
	{
		free(padre);
		free(pfra);
		return NULL;
	}
}
else // ci sono due fratelli
{
	Indice_nodo indice1 = indice + 1;
	Indice_nodo indice2 = indice - 1;
	Tpuntnodo pfra1 = NULL;
	Tpuntnodo pfra2 = NULL;
	if (padre->info[indice1].chiave >= 0)
		pfra1 = carica_nodo(padre->info[indice1].nptr);
	if (padre->info[indice2].chiave >= 0)
		pfra2 = carica_nodo(padre->info[indice2].nptr);

	if ((pfra1 != NULL) && (pfra2 != NULL))
	{
		if ((conta_chiavi_nodo(pfra1) >= LIMITE_INF + 1)
		     && (conta_chiavi_nodo(pfra2) >= LIMITE_INF + 1))
		{
			time_t t;
			srand((unsigned) time(&t));
			Boolean ret = rand() % 2;
			if (ret)
			{
				free(padre);
				free(pfra2);
				return pfra1;
			}
			else
			{
				free(padre);
				free(pfra1);
				return pfra2;
			}
		}
		else if (conta_chiavi_nodo(pfra1) >= LIMITE_INF + 1)
		{
			free(padre);
			free(pfra2);
			return pfra1;
		}
		else if (conta_chiavi_nodo(pfra2) >= LIMITE_INF + 1)
		{
			free(padre);
			free(pfra1);
			return pfra2;
		}
		else
		{
			free(padre);
			free(pfra1);
			free(pfra2);
			return NULL;
		}
	}
	else if (pfra1 != NULL)
	{
		if (conta_chiavi_nodo(pfra1) >= LIMITE_INF + 1)
		{
			free(padre);
			return pfra1;
		}
```

```
                        else
                        {
                                free(padre);
                                free(pfra1);
                                return NULL;
                        }
                }
                else if (pfra2 != NULL)
                {
                        if (conta_chiavi_nodo(pfra2) >= LIMITE_INF + 1)
                        {
                                free(padre);
                                return pfra2;
                        }
                        else
                        {
                                free(padre);
                                free(pfra2);
                                return NULL;
                        }
                }
                else
                {
                        printf("\n Errore: ridistribuzione cancellazione");
                        getch();
                        return NULL;
                }
        }
}

/***************** PROCEDURE VARIE DI CANCELLAZIONE DI UNA CHIAVE ******************/

void cancella_sopra_limite(Tchiave chiave, Tpuntnodo punt, Tnumpagina npagar)
{
        cancella_dal_nodo(chiave, punt);

        Tpuntpagina ppag;
        if (punt->int_ter)
        {
                ppag = carica_pagina(npagar);
                cancella_dalla_pagina(chiave, ppag);
                salva_pagina(ppag, npagar);
                informazioni_archivio_cancellazioni(npagar);
        }
        salva_nodo(punt, punt->np);
}

// cancellazione dalla radice al di sotto del limite inferiore di chiavi

void cancella_dalla_radice(Tchiave chiave, Tpuntnodo pradice, Tnumpagina npagarc)
{
        int nchiavi = conta_chiavi_nodo(pradice);
        Tpuntpagina ppag;
        if (pradice->int_ter)
                ppag = carica_pagina(npagarc);

        if (nchiavi == 2) // radice con una sola chiave: l'altra è INFINITO
        {
                if (pradice->int_ter) // radice terminale con due chiavi
                {
                        printf("\n");
                        printf("\n Cancellazione dell'unica chiave dell'archivio.");
                        printf("\n Adesso l'archivio è vuoto.\n");

                        Tnumpagina pagina_radice = pradice->np;
                        inizializza_nodo(pradice);
                        salva_nodo(pradice, pagina_radice);
                        inizializza_pagina(ppag);
                        salva_pagina(ppag, npagarc);

                        informazioni.indice.pagina_radice = 0L;
                        informazioni.indice.pagina_sequence_set = 0L;
                        informazioni.indice.pagina_vuota_infondo = 0L;
                        for (int i = 0; i < MAXDEL; ++i)
```

```
                                informazioni.indice.pagine_cancellate[i] = 0L;
                        informazioni.archivio.pagina_nonpiena_ultima = 0L;
                        for (i = 0; i < MAXDEL; ++i)
                                informazioni.archivio.pagine_cancellazioni[i] = 0L;
                }
                else // radice intermedia con due chiavi
                {
                        informazioni.indice.pagina_radice = pradice->info[1].nptr;
                        Tnumpagina pagina_radice = pradice->np;
                        inizializza_nodo(pradice);
                        salva_nodo(pradice, pagina_radice);
                        informazioni_indice_cancellate(pagina_radice);
                }
        }
        else // radice sotto il limite ma con più di 2 chiavi
        {
                cancella_dal_nodo(chiave, pradice);
                if (pradice->int_ter)
                {
                        cancella_dalla_pagina(chiave, ppag);
                        salva_pagina(ppag, npagarc);
                        informazioni_archivio_cancellazioni(npagarc);
                }
                salva_nodo(pradice, pradice->np);
        }
}

void cancella_ridistribuzione(Tchiave chiave, Tpuntnodo pp, Tnumpagina npagar,
                                                                Tpuntnodo pfratello)
{
        Tpuntnodo ppadre = carica_nodo(pop());
        push(ppadre->np);

        cancella_dal_nodo(chiave, pp);

        Vet_rid vet;
        for (int i = 0; i < 2 * ORDINE_M; ++i)
        {
                vet[i].chiave = -1L;
                vet[i].nptr = 0L;
        }
        for (i = 0; i < ORDINE_M; ++i)
                if ((pp->info[i].chiave) != -1L)
                        {
                                vet[i].chiave = pp->info[i].chiave;
                                vet[i].nptr = pp->info[i].nptr;
                        }
        for (i = 0; i < ORDINE_M; ++i)
                if ((pfratello->info[i].chiave) != -1L)
                        inserisci_ord_nel_buffer_rid(pfratello->info[i].chiave, vet,
                                                pfratello->info[i].nptr, 2 * ORDINE_M);

        Boolean destra = (pp->info[0].chiave < pfratello->info[0].chiave);

        if (destra)
        {
                for (i = 0; i < LIMITE_INF; ++i)
                {
                        pp->info[i].chiave = vet[i].chiave;
                        pp->info[i].nptr = vet[i].nptr;
                }
                int j;
                for (j = 0, i = LIMITE_INF;
                        (i < 2 * ORDINE_M) && (j < ORDINE_M); ++j, ++i)
                {
                        pfratello->info[j].chiave = vet[i].chiave;
                        pfratello->info[j].nptr = vet[i].nptr;
                }
                ppadre->info[cerca(ppadre, pp->np)].chiave
                        = pp->info[LIMITE_INF - 1].chiave;
        }
        else
        {
                for (i = 0; i < LIMITE_INF; ++i)
```

```
                {
                        pfratello->info[i].chiave = vet[i].chiave;
                        pfratello->info[i].nptr = vet[i].nptr;
                }
                for (i = LIMITE_INF; i < ORDINE_M; ++i)
                {
                        pfratello->info[i].chiave = -1L;
                        pfratello->info[i].nptr = 0L;
                }
                int j;
                for (j = 0, i = LIMITE_INF;
                        (i < 2 * ORDINE_M) && (j < ORDINE_M); ++j, ++i)
                {
                        pp->info[j].chiave = vet[i].chiave;
                        pp->info[j].nptr = vet[i].nptr;
                }
                ppadre->info[cerca(ppadre, pfratello->np)].chiave
                        = pfratello->info[LIMITE_INF - 1].chiave;

        Tpuntpagina ppag;
        if (pp->int_ter)
        {
                ppag = carica_pagina(npagar);
                cancella_dalla_pagina(chiave, ppag);
                salva_pagina(ppag, npagar);
                informazioni_archivio_cancellazioni(npagar);
        }
        salva_nodo(pp, pp->np);
        salva_nodo(ppadre, ppadre->np);
        salva_nodo(pfratello, pfratello->np);
}

// nella seguente procedura di concatenazione:
// se il fratello si trova a destra, viene cancellato il nodo puntato da *pp
// se il fratello si trova a sinistra, viene cancellato tale fratello

void cancella_concatenazione(Tchiave *chiave, Tpuntnodo *pp, Tnumpagina npagar)
{
        Tpuntnodo pfratello = fratello_adiacente(*pp);

        clrscr();
        printf(" Stampa del nodo con cui concatenare le chiavi\n");
        getch();
        stampa_nodo(pfratello);
        getch();

        Tpuntnodo ppadre = carica_nodo(pop());
        push(ppadre->np);

        cancella_dal_nodo((*chiave), (*pp));

        Boolean destra = ((*pp)->info[0].chiave < pfratello->info[0].chiave);

        int i;
        if (destra) // scandisco *pp copiandolo in pfratello
                for (i = 0; i < LIMITE_INF - 1; ++i)
                        inserisci_ord_nel_nodo((*pp)->info[i].chiave, pfratello,
                                                        (*pp)->info[i].nptr);
        else // scandisco pfratello copiandolo in *pp
                for (i = 0; i < LIMITE_INF; ++i)
                        inserisci_ord_nel_nodo(pfratello->info[i].chiave, (*pp),
                                                    pfratello->info[i].nptr);

        Tpuntpagina ppag;
        if ((*pp)->int_ter)
        {
                ppag = carica_pagina(npagar);
                cancella_dalla_pagina((*chiave), ppag);
                salva_pagina(ppag, npagar);
                informazioni_archivio_cancellazioni(npagar);

                if (destra)
                {
                        pfratello->npp = (*pp)->npp;
```

```
                        if ((*pp)->npp != 0L)
                        {
                                Tpuntnodo pp1 = carica_nodo((*pp)->npp);
                                pp1->nps = (*pp)->nps;
                                salva_nodo(pp1, pp1->np);
                        }
                }
                else
                {
                        (*pp)->npp = pfratello->npp;
                        if (pfratello->npp != 0L)
                        {
                                Tpuntnodo pp1 = carica_nodo(pfratello->npp);
                                pp1->nps = pfratello->nps;
                                salva_nodo(pp1, pp1->np);
                        }
                }
        }

        if (destra)
        {
                (*chiave) = ppadre->info[cerca(ppadre, (*pp)->np)].chiave;

                Tnumpagina pag_pp = (*pp)->np;
                informazioni_indice_cancellate(pag_pp);

                if (pag_pp == informazioni.indice.pagina_sequence_set)
                        informazioni.indice.pagina_sequence_set = pfratello->np;

                inizializza_nodo((*pp));
                salva_nodo((*pp), pag_pp);
                salva_nodo(pfratello, pfratello->np);
        }
        else
        {
                (*chiave) = ppadre->info[cerca(ppadre, pfratello->np)].chiave;

                Tnumpagina pag_fra = pfratello->np;
                informazioni_indice_cancellate(pag_fra);

                if (pag_fra == informazioni.indice.pagina_sequence_set)
                        informazioni.indice.pagina_sequence_set = (*pp)->np;

                inizializza_nodo(pfratello);
                salva_nodo(pfratello, pag_fra);
                salva_nodo((*pp), (*pp)->np);
        }

        salva_nodo(ppadre, ppadre->np);
        (*pp) = carica_nodo(pop());
}

/************************** CANCELLAZIONE DI UNA CHIAVE **************************/

/* cancellazione di una chiave presente in un archivio già creato */

void cancellazione(Tchiave chiave, Tnumpagina npagar)
{
        Tpuntnodo pp = carica_nodo(pop()); // pp punta al nodo in cui cancellare
        Boolean termine = 0;

        while (!termine)
        {

        clrscr();
        printf(" Stampa del nodo in cui va cancellata la chiave %ld\n", chiave);
        getch();
        stampa_nodo(pp);
        getch();

        if (conta_chiavi_nodo(pp) >= LIMITE_INF + 1) /* siamo sopra il limite */
        {
                cancella_sopra_limite(chiave, pp, npagar);
                termine = 1;
```

```
        }
        else /* siamo sotto il limite */
        {
                if (pp->np == informazioni.indice.pagina_radice)
                {                                       /* radice sotto il limite */
                        clrscr();
                        printf("\n\n\n Cancellazione dalla radice con un numero");
                        printf(" di chiavi\n al di sotto del limite inferiore\n");
                        getch();

                        cancella_dalla_radice(chiave, pp, npagar);
                        termine = 1;
                }
                else /* nodo diverso dalla radice sotto il limite */
                {
                        Tpuntnodo pfrat = ridistribuzione_can(pp);
                        if (pfrat != NULL) /* ridistribuzione */
                        {
                                clrscr();
                                printf("\n\n\n Cancellazione con ridistribuzione\n");
                                getch();
                                clrscr();
                                printf(" Stampa del nodo");
                                printf(" con cui ridistribuire le chiavi\n");
                                getch();
                                stampa_nodo(pfrat);
                                getch();

                                cancella_ridistribuzione(chiave, pp, npagar, pfrat);
                                termine = 1;
                        }
                        else /* concatenazione */
                        {
                                clrscr();
                                printf("\n\n\n Cancellazione con concatenazione\n");
                                getch();

                                cancella_concatenazione(&chiave, &pp, npagar);
                        }
                }
        }

        } /* fine del while (!termine) */
}
/************************** CANCELLAZIONE DI UNA CHIAVE **************************/

void cancella_chiave(Tchiave chiave, Tnumpagina pag_radice)
{
        clrscr();
        printf("\n PROCEDURA DI CANCELLAZIONE DI UNA CHIAVE.\n");
        printf("\n Cancellazione della chiave %ld.\n", chiave);
        getch();

        Tnumpagina npagarc;

        if ((npagarc = ricerca_chiave(chiave, pag_radice)) == -1L)
        {
                printf(" e quindi è impossibile cancellare qualsiasi chiave.\n");
                getch();
        }
        else if (npagarc == 0L)
        {
                printf("\n Chiave non presente: impossibile cancellarla.\n");
                getch();
        }
        else
                cancellazione(chiave, npagarc);
}
```

```
/******** FUNZIONE CHE MODIFICA LE INFORMAZIONI DI UNA CHIAVE DELL'ARCHIVIO ********/

void modifica_dati(Tchiave chiave, Tnumpagina radice, char c[20], char n[20])
{
	clrscr();
	printf("\n PROCEDURA DI MODIFICA DELL'INFORMAZIONE DI UNA CHIAVE.\n");
	printf("\n Modifica della chiave %ld.\n", chiave);
	getch();

	Tnumpagina npagarc;

	if ((npagarc = ricerca_chiave(chiave, radice)) == -1L)
	{
		printf(" e quindi è impossibile modificare le informazioni.\n");
		getch();
	}
	else if (npagarc == 0L)
	{
		printf("\n Chiave non presente:");
		printf(" impossibile modificare le relative informazioni.\n");
		getch();
	}
	else
	{
		Tpuntpagina ppag = carica_pagina(npagarc);

		int i = 0;
		while ((*ppag)[i].chiave != chiave)
			++i;

		strcpy((*ppag)[i].cognome, c);
		strcpy((*ppag)[i].nome, n);

		salva_pagina(ppag, npagarc);

		printf("\n Modifica effettuata con successo.\n");
		getch();
	}
}

/*************************** STAMPA DELL'ALBERO ***************************/

#define MAXLIVELLO 5000

void inizializza_vettore(Tnumpagina vet[])
{
	for (int i = 0; i < MAXLIVELLO; ++i)
		vet[i] = 0L;
}

void copia_vettori(Tnumpagina figli_1_[], Tnumpagina figli_2_[])
{
		for (int i = 0; i < MAXLIVELLO; ++i)
			figli_1_[i] = figli_2_[i];
}

void stampa_chiavi_nodo(Tpuntnodo pnodo)
{
	printf("▌");
	for (int i = 0; (i < ORDINE_M) && (pnodo->info[i].chiave != -1L); ++i)
		if (pnodo->info[i].chiave == INFINITO)
			printf(" ∞");
		else
			printf(" %ld", pnodo->info[i].chiave);
	printf(" ▌ ");
}
```

```
void stampa_albero(void)
{
	clrscr();
	printf("\nSTAMPA DELL'ALBERO\n\n");

	Tnumpagina figli_1_[MAXLIVELLO];
	Tnumpagina figli_2_[MAXLIVELLO];

	inizializza_vettore(figli_1_);

	figli_1_[0] = informazioni.indice.pagina_radice;

	int i, j, k;
	Tpuntnodo pnodo;
	Boolean finito = 0;

	while (!finito)
	{
		inizializza_vettore(figli_2_);
		for (i = 0; i < MAXLIVELLO; ++i)
		{
			if (figli_1_[i] != 0L)
			{
				pnodo = carica_nodo(figli_1_[i]);
				stampa_chiavi_nodo(pnodo);
				for (k = i * ORDINE_M, j = 0; j < ORDINE_M; ++j)
				{
					if (k > MAXLIVELLO)
					{
						printf("\n Errore: aumentare");
						printf(" MAXLIVELLO");
						getch();
						exit(27);
					}
					if (pnodo->int_ter)
						finito = 1;
					else
					{
						if (pnodo->info[j].nptr != 0L)
							figli_2_[k++] =
								pnodo->info[j].nptr;

					}
				}
				free(pnodo);
			}
		}
		printf("\n\n");
		copia_vettori(figli_1_, figli_2_);
	}
	getch();
}
```