13. ARBORI B

13.1 Arbori B. Definiţie. Proprietăţi.

În cazul sistemelor de gestiune a bazelor de date relaţionale (SGBDR) este important ca pe lângă stocarea datelor să se realizeze şi regăsirea rapidă a acestora. În acest scop sunt folosiţi indecşii. Un *index* este o colecţie de perechi <*valoare cheie, adresa articol*>. Scopul primar al unui index este acela de a facilita accesul la o colecţie de articole. Un index se spune ca este *dens* dacă el conţine câte o pereche <*valoare cheie, adresa articol*> pentru fiecare articol din colecţie. Un index care nu este dens uneori este numit *index rar*.

Structura de date foarte des folosită pentru implementarea indecşilor este *arborele de căutare*. Articolele memorate pot fi oricât de complexe, dar ele conţin un câmp numit *cheie* ce serveşte la identificarea acestora. Să notăm cu C mulţimea cheilor posibile ce vor trebui regăsite cu ajutorul arborelui de căutare. Dacă arborele de căutare este astfel construit încât foloseşte o relaţie de ordine totală pe C, atunci vom spune că arborele de căutare este bazat pe ordinea cheilor. Arborii de căutare, bazaţi pe ordinea cheilor, sunt de două feluri: *arbori binari de căutare* (au o singură cheie asociată fiecărui nod) sau *arbori multicăi de căutare* (au mai multe chei asociate fiecărui nod).

Performanţele unui index se îmbunătăţesc în mod semnificativ prin mărirea factorului de ramificare a arborelui de căutare folosit. *Arborii multicăi de căutare* sunt o generalizare a arborilor binari de căutare. Astfel, unui nod oarecare, în loc să i se ataşeze o singură cheie care permite ramificarea în doi subarbori, i se ataşează un număr de *m* chei, ordonate strict crescător, care permit ramificarea în *m + 1* subarbori. Numărul *m* diferă de la nod la nod, dar în general pentru fiecare nod trebuie să fie între anumite limite (ceea ce va asigura folosirea eficientă a mediului de stocare). Cele *m* chei ataşate unui nod formează o *pagină*. Determinarea poziţiei cheii căutate în cadrul unui nod se realizează secvenţial în cazul paginilor cu număr mic de chei sau prin căutare binară. Un exemplu de arbore multicăi de căutare de ordin 3 este dat în figura 13.1.

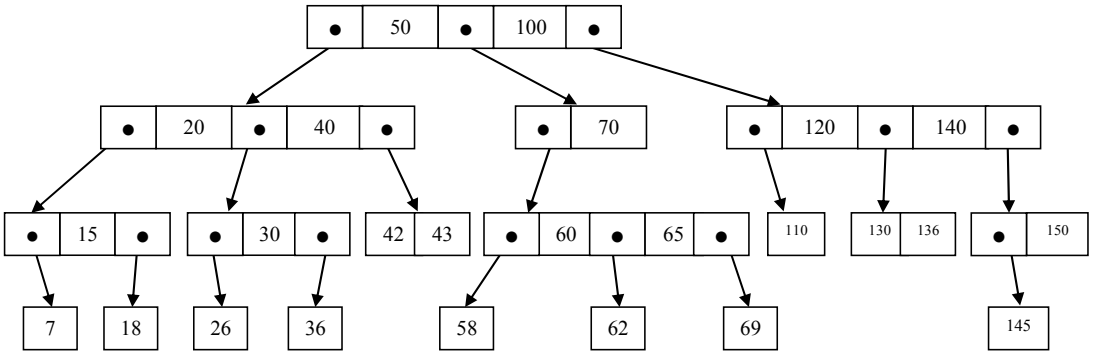

Figura 13.1 *Arbore multicăi de căutare de ordin 3*

Un arbore multicăi de căutare care nu este vid are următoarele proprietăţi:

- fiecare nod al arborelui are structura dată în figura 13.2;

n	P_0	K_0	P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

Figura 13.2 *Structura de nod pentru un arbore multicăi de căutare de ordin n*

unde P_0, P_1, …, P_n sunt pointeri către subarbori şi K_0, K_1, … , K_{n-1} sunt valorile cheilor. Cerinţa ca fiecare nod să aibă un număr de ramificaţii mai mic sau egal decât *m* conduce la restricţia $n \leq m - 1$.

- valorile cheilor într-un nod sunt date în ordine crescătoare:

$$K_i < K_{i+1}, \quad i = \overline{0, n-2} \qquad (13.1)$$

- toate valorile de chei din nodurile subarborelui indicat de P_i sunt mai mici decât valoarea cheii K_i, $i = \overline{0, n-1}$.
- toate valorile de chei din nodurile subarborelui indicat de P_n sunt mai mari decât valoarea de cheie K_{n-1}.
- subarborii indicaţi de P_i, $i = \overline{0, n}$ sunt de asemenea arbori multicăi de căutare.

Prima oară arborele B a fost descris de R. Bayer şi E. McCreight în 1972. Arborii B rezolvă problemele majore întâlnite la implementarea arborilor de căutare stocaţi pe disc:

- au întotdeauna toate nodurile frunză pe acelaşi nivel (cu alte cuvinte sunt echilibraţi după înălţime);
- operaţiile de căutare şi actualizare afectează puţin blocuri pe disc;
- păstrează articolele asemănătoare în acelaşi bloc pe disc;
- garantează ca fiecare nod din arbore va fi plin cu un procent minim garantat.

Un arbore B de ordin *m* este un arbore multicăi de căutare şi are următoarele proprietăţi:

i. toate nodurile frunză sunt pe acelaşi nivel;
ii. rădăcina are cel puţin doi descendenţi, dacă nu este frunză;
iii. fiecare pagină conţine cel puţin $\left\lceil \frac{m}{2} \right\rceil$ chei (excepţie face rădăcina care poate avea mai puţine chei, dacă este frunză);
iv. nodul este fie frunză, fie are n + 1 descendenţi (unde n este numărul de chei din nodul respectiv, cu $\left\lceil \frac{m}{2} \right\rceil \leq n \leq m-1$);
v. fiecare pagină conţine cel mult *m-1* chei; din acest motiv, un nod poate avea maxim *m* descendenţi.

Proprietatea *i* menţine arborele balansat. Proprietatea *ii* forţează arborele să se ramifice devreme. Proprietatea *iii* ne asigură că fiecare nod al arborelui este cel puţin pe jumătate plin.

Înălţimea maximă a unui arbore B dă marginea superioară a numărului de accese la disc necesare pentru a localiza o cheie.

Se consideră *h* înălţimea maximă a unui arbore B cu *N* chei, unde valoarea indicatorului este dată de relaţia:

$$h = \log_{\left\lceil \frac{m}{2} \right\rceil}\left(\frac{N+1}{2}\right) \qquad (13.2)$$

Ca exemplu, pentru N = 2.000.000 şi m = 20, înălţimea maximă a unui arbore B de ordin m va fi 3, pe când un arborele binar corespondent va avea o înălţime mai mare de 20.

13.2 Operaţii de bază într-un arbore B

Procesul de *căutare* într-un arbore B este o extindere a căutării într-un arbore binar. Operaţia de căutare în arborele B se realizează comparând cheia căutată *x* cu cheile nodului curent, plecând de la nodul rădăcină. Dacă nodul curent are *n* chei, atunci se disting următoarele cazuri:

- $c_i < x < c_{i+1}$, $i = \overline{1,n}$ – se continuă căutarea în nodul indicat de P_i;
- $c_n < x$ – se continuă căutarea în nodul indicat de P_n;
- $x < c_0$ – se continuă căutarea în nodul indicat de P_0.

Lungimea maximă a drumului de căutare este dată de înălţimea arborelui. Fiecare referire a unui nod implică selecţia unui subarbore.

Arborele B suportă *căutarea secvenţială* a cheilor. Arborele este traversat secvenţial prin referirea în inordine a nodurilor. Un nod este referit de mai multe ori întrucât el conţine mai multe chei. Subarborele asociat fiecărei chei este referit înainte ca următoarea cheie sa fie accesata. Arborii B sunt optimi pentru accesul direct la o cheie. Pentru accesul secvenţial la o cheie nu se obţin performanţe satisfăcătoare.

Condiţia ca toate frunzele să fie pe acelaşi nivel duce la un comportament caracteristic arborilor B: faţă de arborii binari de căutare, arborilor B nu le este permis să crească la frunze; ei sunt forţaţi sa crească la rădăcină.

Operaţia de *inserare* a unei chei în arborele B este precedată de operaţia de căutare. În cazul unei căutări cu succes (cheia a fost găsită în arbore) nu se mai pune problema inserării întrucât cheia se afla deja în arbore. Dacă cheia nu a fost găsită, operaţia de căutare se va termina într-un nod frunză. În acest nod frunză se va insera noua cheie. Funcţie de gradul de umplere al nodului frunză afectat, se disting următoarele cazuri:

- nodul are mai puţin de *m – 1* chei; inserarea se efectuează fără să se modifice structura arborelui ;
- nodul are deja numărul maxim de *m – 1* chei; în urma inserării nodul va avea prea multe chei, de aceea el va "fisiona". În urma fisionării vom obţine două noduri care se vor găsi pe acelaşi nivel şi o cheie mediană care nu se va mai găsi în nici unul din cele două noduri. Cele $\left\lceil \frac{m}{2} \right\rceil$ chei din stânga rămân în nodul care fisionează. Cele $\left\lceil \frac{m}{2} \right\rceil$ din dreapta vor forma cel de-al doilea nod. Cheia mediană va urca în nodul părinte, care la rândul lui poate să

Se observă că procesul de inserare a unei chei garantează că fiecare nod intern va avea cel puţin jumătate din numărul maxim de descendenţi. În urma operaţiilor de inserare arborele va deveni mai înalt şi mai lat, figura 13.3.

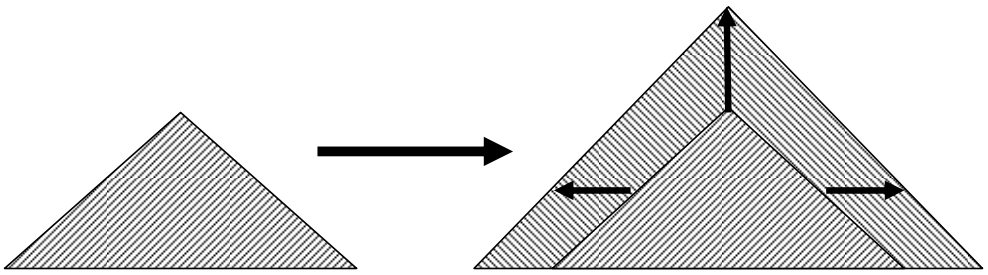

Figura 13.3 *Modificările dimensionale ale unui arbore B după efectuarea inserării*

Să considerăm un arbore B de ordin 5 (deci numărul maxim de chei dintr-un nod va fi 4). În urma inserării valorilor de cheie 22, 57, 41, 59 nodul rădăcină va fi cel din figura 13.4.

Figura 13.4 *Structura nodului rădăcină după inserarea cheilor*

În urma inserării cheii 54, nodul rădăcină va conţine prea multe chei, aşa că el va fisiona, figura 13.5.

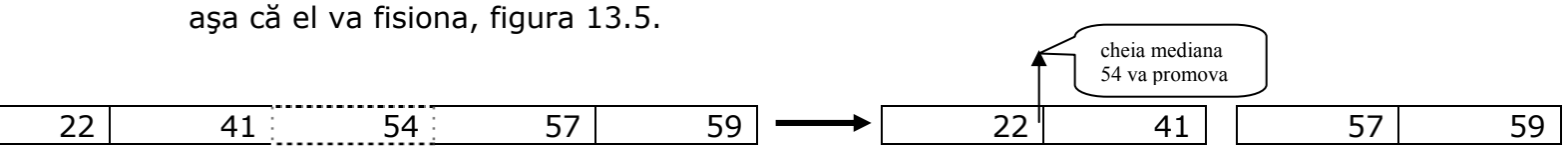

Figura 13.5 *Fisionarea nodului rădăcină*

În urma promovării cheii mediene 54 se va forma o nouă rădăcină, arborele crescând în înălţime cu 1, figura 13.6.

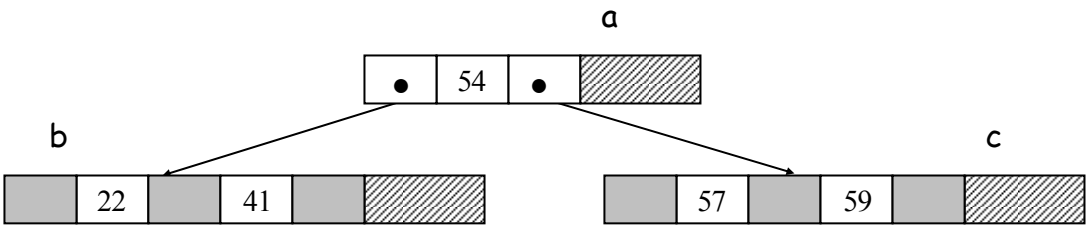

Figura 13.6 *Formarea noii rădăcini a arborelui*

Inserarea cheilor 33, 75, 124 nu ridică probleme, figura 13.7.

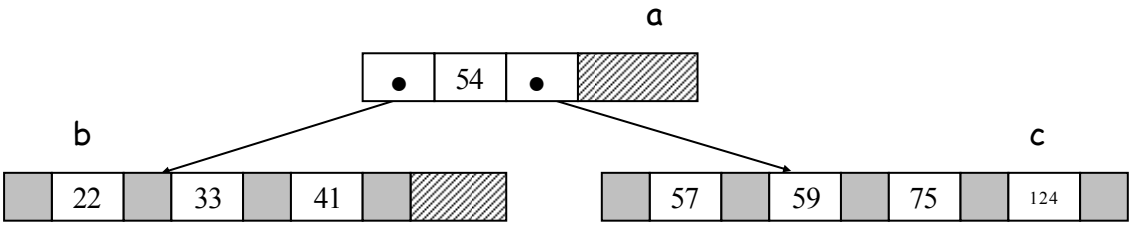

Figura 13.7 *Structura arborelui după inserarea cheilor 33, 75, 124*

Inserarea cheii 62 însă duce la divizarea nodului *c*, figura 13.8.

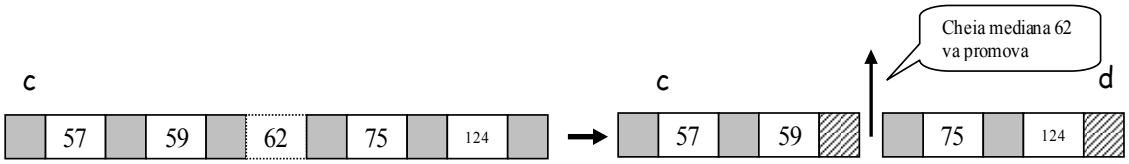

Figura 13.8 *Fisionarea nodului c*

Cheia 62 va promova în nodul rădăcină, figura 13.9.

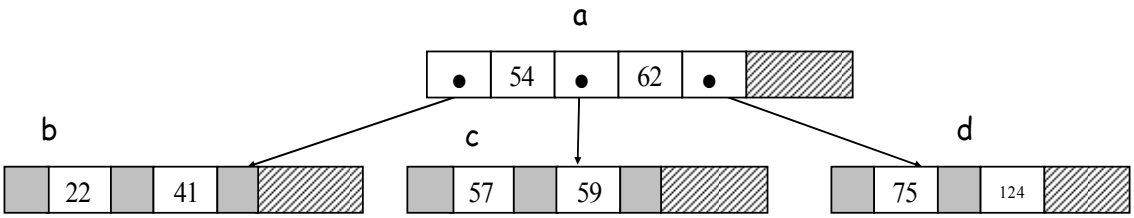

Figura 13.9 *Promovarea cheii 62 în rădăcină*

În urma inserării cheilor 33, 122, 123, 55, 60, 45, 66, 35 configuraţia arborelui va fi cea din figura 13.10.

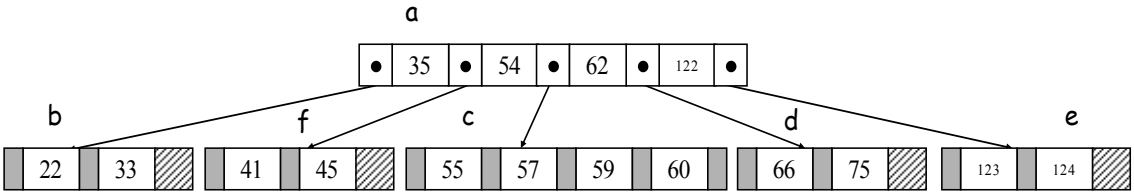

Figura 13.10 *Structura arborelui după inserări succesive*

Inserarea cheii 56 se va face în nodul *c* şi acesta va fisiona, figura 13.11.

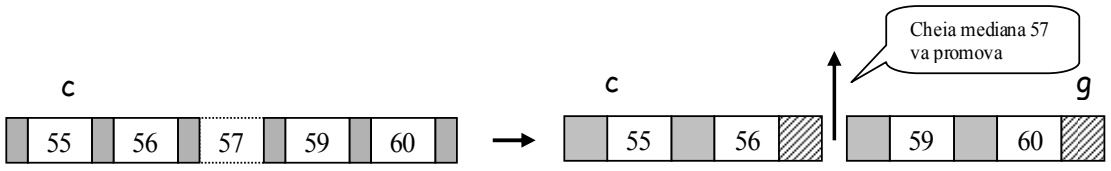

Figura 13.11 *Fisionarea nodului c*

Oricum, nodul părinte a este deja plin şi nu poate primi noua cheie 57 şi pointerul către nodul nou format f. Algoritmul de fisionare este aplicat din nou, dar de data aceasta nodului *a*, figura 13.12.

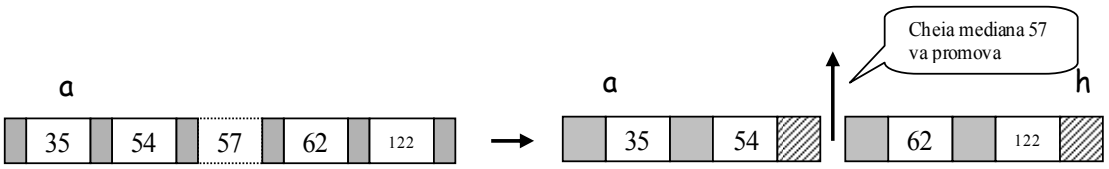

Figura 13.12 *Fisionarea nodului a*

Cheia mediană promovată 57 va forma noua rădăcină a arborelui B, având subarborele stâng *a* şi subarborele drept *h*, figura 13.13.

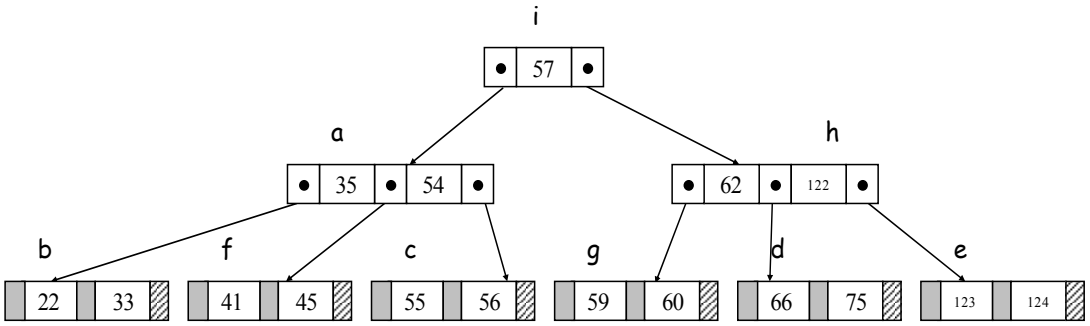

Figura 13.13 *Noua configuraţie a arborelui B*

Uzual, în special pentru arbori B de ordin mare, un nod părinte are suficient spaţiu disponibil pentru a primi valoarea unei chei şi un pointer către un nod descendent. În cel mai rău caz algoritmul de fisionare este aplicat pe întreaga înălţime a arborelui. În acest mod arborele va creşte în înălţime, lungimea drumului de căutare crescând cu 1.

Sintetizat, algoritmul de inserare a unei valori de cheie în arbore este prezentat în figura 13.14.

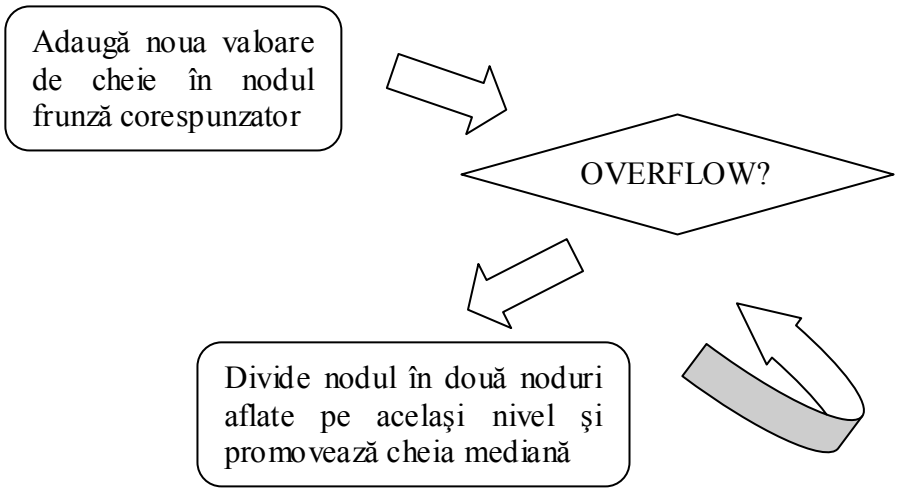

Figura 13.14 *Algoritmul de inserare a unei chei în arbore*

Algoritmul de inserare într-un arbore B (pseudocod):

- inserează noua valoare de cheie în nodul frunză corespunzător;
- nodul_curent = nodul_frunza;
- while(starea pentru nodul_curent este *OVERFLOW*):
  - divide nodul_curent în două noduri aflate pe acelaşi nivel şi promovează cheia mediană în nodul părinte pentru nodul_curent;
  - nodul_curent = nodul părinte pentru nodul_curent.

În cel mai rău caz, inserarea unei chei noi duce la aplicarea algoritmului de fisionare pe întreaga înălţime a arborelui, fisionându-se *h – 1* noduri, unde *h* este înălţimea arborelui înainte de inserare. Numărul total de fisionări care au apărut când arborele avea *p* noduri este de *p – 2*. Prima fisionare adaugă două noduri noi; toate celelalte fisionări produc doar un singur nod nou. Fiecare nod are un minim de $\left\lceil \frac{m}{2} \right\rceil - 1$ chei, cu excepţia rădăcinii, care are cel puţin o cheie. Deci arborele cu *p* noduri conţine cel puţin $1 + (p-1)\left(\left\lceil \frac{m}{2} \right\rceil - 1\right)$ chei. Probabilitatea ca o fisionare să fie necesară după inserarea unei valori de cheie este mai mică decât :

$$\frac{p-2}{1+(p-1)\left(\left\lceil \frac{m}{2} \right\rceil - 1\right)} \qquad \frac{(divizari)}{chei} \qquad (13.3)$$

care este mai mică decât 1 divizare per $\left\lceil \frac{m}{2} \right\rceil - 1$ inserări de chei (deoarece $\frac{1}{p-2}$ tinde către 0 pentru o valoare mare a lui *p* şi $\frac{p-1}{p-2}$ este aproximativ 1). De exemplu, pentru m = 10, probabilitatea apariţiei unei divizări este de 0.25. Pentru m = 100, probabilitatea divizării este 0.0204. Pentru m = 200, probabilitatea divizării este 0.0101. Cu alte cuvinte, cu cât ordinul arborelui este mai mare, cu atât este mai mică probabilitatea ca inserarea unei valori de cheie sa ducă la divizarea unui nod.

Operaţia de *ştergere* dintr-un arbore B este ceva mai complicată decât operaţia de inserare. Operaţia de ştergere se realizează simplu dacă valoarea de cheie care urmează a fi ştearsă se află într-un nod frunză. Dacă nu, cheia va fi ştearsă logic, fiind înlocuită cu o alta, vecină în inordine, care va fi ştearsă efectiv. În urma ştergerii se disting următoarele cazuri:

- dacă nodul conţine mai mult de $\left\lceil \frac{m}{2} \right\rceil$ chei, ştergerea nu ridică probleme;
- dacă nodul are numărul minim de chei $\left\lceil \frac{m}{2} \right\rceil$, după ştergere numărul de chei din nod va fi insuficient. De aceea se împrumută o cheie din nodul vecin (aflat pe acelaşi nivel în arbore) dacă acesta are cel puţin $\left\lceil \frac{m}{2} \right\rceil$ chei, caz în care avem de-a face cu o *partajare*. Dacă nu se poate face o partajare cu nici unul din

Să considerăm următoarea configuraţie de arbore B de ordin 5 din figura 13.15.

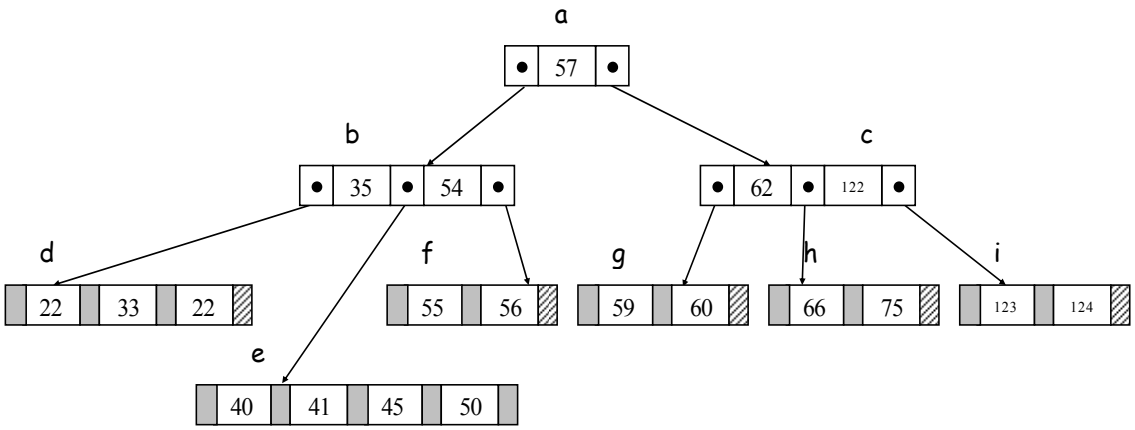

Figura 13.15 *Arbore B de ordin 5*

Ştergerea valorilor de cheie 40 şi 45 din nodul *e* nu ridică probleme, figura 13.16

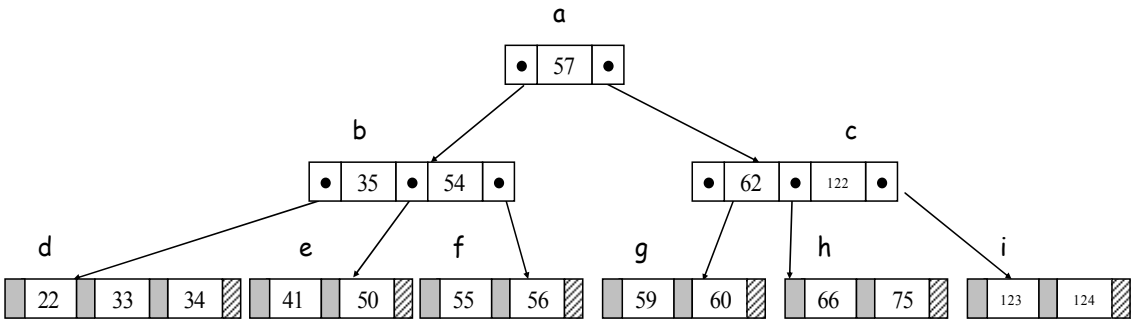

Figura 13.16 *Structura arborelui după ştergerea cheilor 40 şi 45*

Ştergerea valorii de cheie 50 necesită însă partajarea între nodurile *d* şi *e*, figura 13.17.

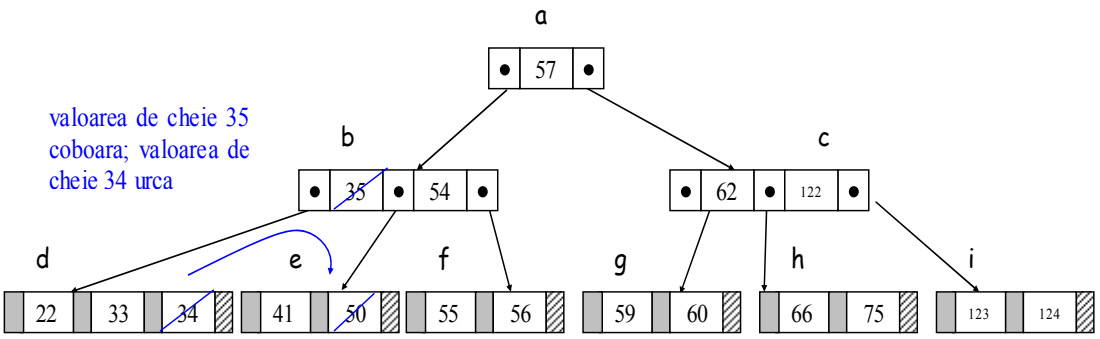

Figura 13.17 *Partajarea între nodurile d şi e*

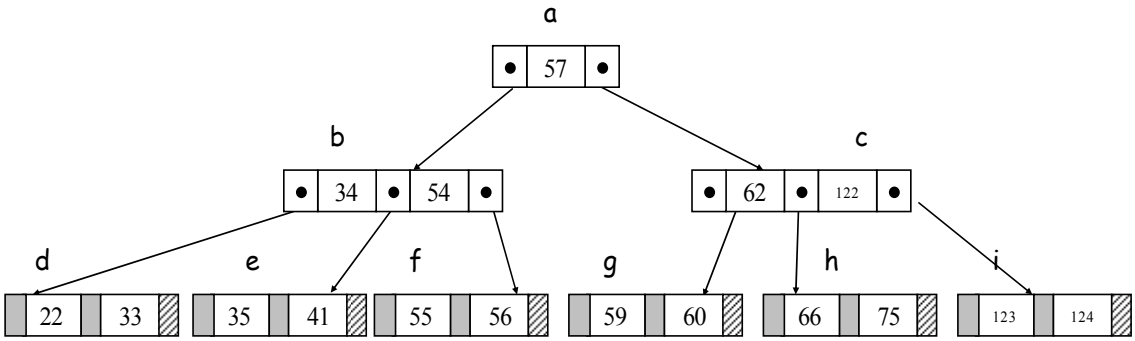

Figura 13.18 *Structura arborelui după partajare*

Ştergerea valorii de cheie 22 va necesita fuzionarea nodurilor *d* şi *e*, figura 13.19.

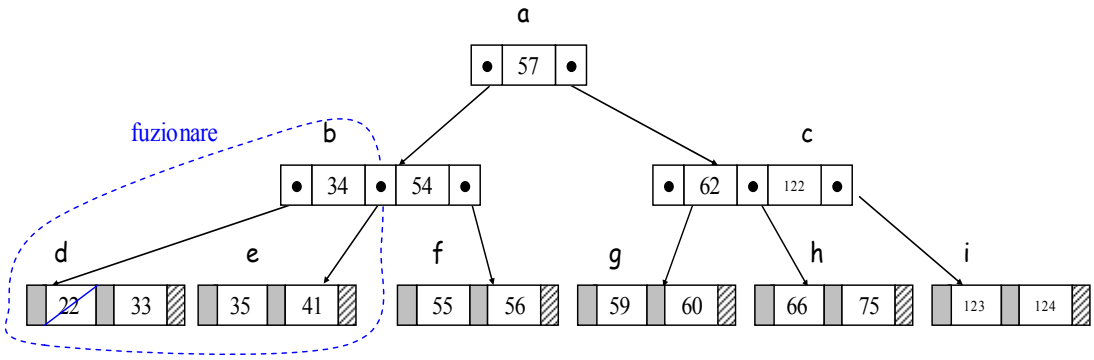

Figura 13.19 *Fuzionarea nodurilor d şi e*

Însă în urma fuzionării nodurilor *d* şi *e*, nodul *b* va conţine prea puţine valori de cheie, aşa că vor fuziona şi nodurile *b* şi *c*, figura 13.20.

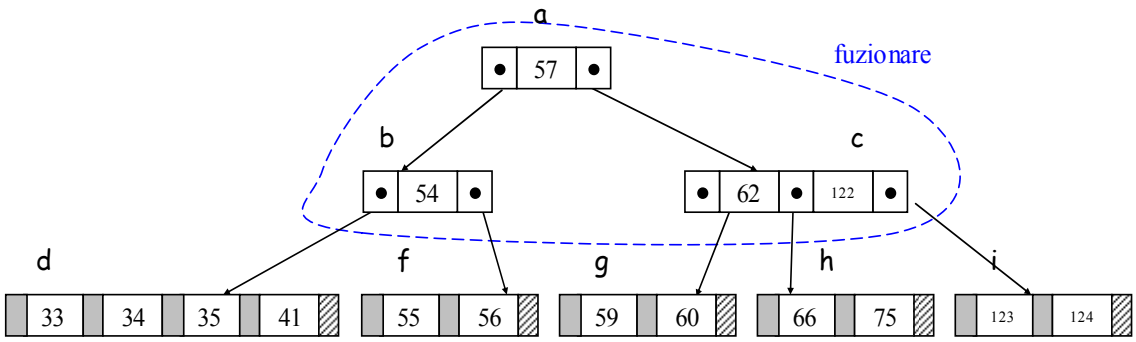

Figura 13.20 *Fuzionarea nodurilor a, b şi c*

Astfel, în final, arborele B va arăta astfel ca în figura 13.21.

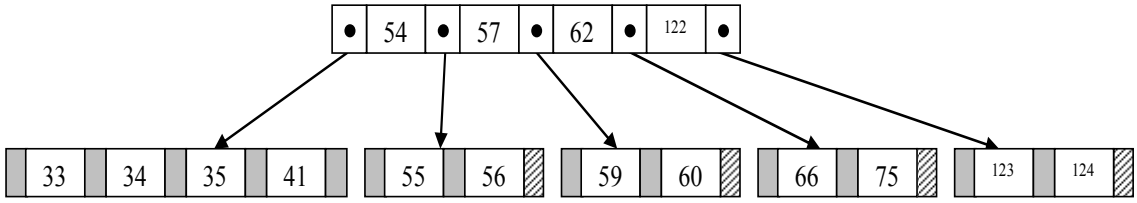

Figura 13.21 *Structura finală a arborelui B*

Algoritmul de ştergere dintr-un arbore B, în pseudocod:

- *if* (valoarea de cheie care se şterge nu este într-un nod frunză) *then:* înlocuieşte valoarea de cheie cu succesor / predecesor;
- nodul_curent = nodul_frunza;
- *while* (starea pentru nodul_curent este *UNDERFLOW*):
  - încearcă partajarea cu unul din nodurile vecine aflate pe acelaşi nivel, via nodul părinte;
  - *if* (nu este posibil) *then*:
    1. fuzionează nodul_curent cu un nod vecin, folosind o valoare de cheie din nodul părinte;
    2. nodul_curent = părintele pentru nodul_curent.

13.3 Algoritmii C++ pentru inserarea unei valori de cheie într-un arbore B

Înainte de a prezenta algoritmii pentru căutare şi inserare într-un arbore B, să începem mai întâi cu declaraţiile necesare pentru implementarea unui arbore. Pentru simplicitate vom construi întregul arbore B în memoria heap, folosind pointeri pentru a descrie structura arborescentă. În majoritatea aplicaţiilor, aceşti pointeri vor trebui înlocuiţi cu adrese către blocuri sau pagini stocate într-un fişier pe disc (deci accesarea unui pointer va însemna un acces la disc).

Un nod al arborelui B va fi descris de clasa *btree_node.* Pentru a fi cât mai generală în ceea ce priveşte tipul valorii de cheie folosită va fi implementată ca fiind o clasă *template*. Oricum, tipul valorii de cheie folosită va trebui să permită ordonarea cheilor; în cazul implementării de

faţă cerinţa este de a suporta testele de inegalitate strictă (<) şi de egalitate (= =).

Pentru generalitate se vor folosi funcţii pentru testele de egalitate şi inegalitate strictă. Declaraţiile acestora sunt următoarele:

```
template <typename FIRST_ARGUMENT_TYPE,
                typename SECOND_ARGUMENT_TYPE,
          typename RETURN_TYPE
       >
struct binary_function { /** empty */ } ;

template <typename TYPE>
struct less_than : public binary_function<TYPE, TYPE, bool> {
  inline bool operator()( const TYPE& first, const TYPE& second ) {
    return first < second ;
  }
} ; /**   struct less_than                                          */

template <>
struct less_than<char*> : public binary_function<char*, char*, bool> {
  inline bool operator()( const char* first, const char* second ) {
    return strcmp( first, second ) < 0 ;
  }
} ; /**   struct less_than<char*>                                   */

template <typename TYPE>
struct equal_to : public binary_function<TYPE, TYPE, bool> {
  inline bool operator()( const TYPE& first, const TYPE& second ) {
    return first == second ;
  }
} ; /**   struct equal_to                                          */

template <>
struct equal_to<char*> : public binary_function<char*, char*, bool> {
  inline bool operator()( const char* first, const char* second ) {
    return strcmp( first, second ) == 0 ;
  }
} ; /**   struct equal_to<char*>                                   */
```

Declaraţia clasei *btree_node* este:

```
template <typename KEY_NODE_TYPE>
class btree_node {
  friend class btree<KEY_NODE_TYPE> ;

public:
  typedef KEY_NODE_TYPE key_node_type ;
  typedef key_node_type* key_node_ptr ;
  typedef btree_node<KEY_NODE_TYPE> btree_node_type ;
  typedef btree_node_type* btree_node_ptr ;

  typedef  struct {
    key_node_type   key_value ;
    btree_node_ptr  node_ptr ;
  } btree_node_tuple ;

public:
  enum BTREE_NODE_STATUS {
    EMPTY = 0,
    UNDERFLOW,
```

```
    MINIMAL,
    OPERATIONAL,
    FULL,
    OVERFLOW
  } ;

public:
  btree_node( int ncapacity ) ;
  virtual ~btree_node( void ) ;

  BTREE_NODE_STATUS status( void ) const ;
  int capacity( void ) const ;
  int size( void ) const ;

  key_node_type& key_value( int position ) ;
  btree_node<KEY_NODE_TYPE>* child( int position ) ;

  int push( const key_node_type& value, btree_node_ptr child ) ;
  bool find( const key_node_type& value, int& position ) ;
  int remove_at( const int position ) ;

  bool is_leaf( void ) const ;

  int split( btree_node_tuple* tuple ) ;

protected:
  static int initialize( btree_node_ptr node ) ;
  int shift2right( const int start ) ;
  int shift2left( const int start ) ;

protected:
  btree_node( const btree_node<KEY_NODE_TYPE>& ) ;
  btree_node<KEY_NODE_TYPE>& operator =( const
btree_node<KEY_NODE_TYPE>& ) ;

protected:
  less_than<key_node_type>  _less_than ;
  equal_to<key_node_type>   _equal_to ;

protected:
  key_node_ptr                node_keys ;
  btree_node_ptr*             node_childs ;
  int                    node_capacity ;
  int                    node_size ;
} ;   /**   class btree_node                    */
```

Structura *btree_node_tuple* va fi folosită de algoritmul de fisionare (divizare) a unui nod. Valorile din cadrul enumerării *BTREE_NODE_STATUS* înseamnă:

- *EMPTY* – nodul nu conţine nici o cheie ;
- *UNDERFLOW* – nodul conţine prea puţine chei ;
- *MINIMAL* – nodul conţine numărul minim de chei ;
- *OPERATIONAL* – numărul de chei conţinut de nod respecta cerinţele de arbore B;
- *FULL* – nodul conţine numărul maxim de chei ;
- *OVERFLOW* – nodul conţine prea multe chei.

Ordinea în care câmpurile din enumerare apar este importantă (a se vedea funcţiile de introducere şi ştergere a unei chei din arbore). Constructorul clasei *btree_node* primeşte ca parametru numărul maxim de

chei care pot fi păstrate într-un nod (capacitatea nodului). Operaţiile de copiere a unui nod nu sunt permise (este dezactivat constructorul de copiere şi operatorul de atribuire). Câmpurile *_less_than* şi *_equal_to* sunt folosite în testele de egalitate şi inegalitate strictă. Câmpul *node_capacity* păstrează numărul maxim de chei dintr-un nod (capacitatea nodului). Câmpul *node_size* menţine numărul de chei aflate la un moment dat într-un nod. Câmpul *node_keys* este un vector în care vor fi păstrate valorile de chei din nod (în implementarea de faţă, pentru implementarea mai uşoara a algoritmului de divizare a unui nod, numărul de chei păstrate într-un nod va fi cu unu mai mare decât numărul maxim de chei). Câmpul *node_childs* va păstra pointeri către descendenţi (numărul de descendenţi ai unui nod este cu unu mai mare decât numărul de chei din acel nod).

Constructorul *btree_node<KEY_TYPE>::btree_node()* apelează funcţia *initialize()* care iniţializează un nod al arborelui (alocă memoria necesară pentru păstrarea valorilor de chei şi a pointerilor către descendenţi, setează numărul de chei prezente în arbore la 0):

```
template <typename KEY_NODE_TYPE>
btree_node<KEY_NODE_TYPE>::btree_node( int capacity ) {
  this->node_capacity = capacity ;
  initialize( this );
}

template <typename KEY_NODE_TYPE>
int
btree_node<KEY_NODE_TYPE>::initialize( btree_node_ptr node ) {
  if( 0 = = node ) return -1 ;
  node->node_keys = new key_node_type[ node->capacity() + 1 ];
  if( 0 = = node->node_keys ) return -1 ;
  node->node_childs = new btree_node_ptr[ node->capacity() + 2 ];
  if( 0 = = node->node_childs ) {
    delete []node->node_keys ;
    node->node_keys = 0;
    return -1 ;
  }
  memset(node->node_childs,0,sizeof(btree_node_ptr)*
(node->capacity() + 2 ) );
  node->node_size = 0;

  return 0;
}
```

Destructorul *btree_node<KEY_TYPE>::~btree_node()* eliberează memoria alocată:

```
template <typename KEY_NODE_TYPE>
btree_node<KEY_NODE_TYPE>::~btree_node( void ) {
  delete []this->node_keys ;
  delete []this->node_childs ;
}
```

Pentru a se afla numărul de chei aflate la un moment dat într-un nod se foloseşte funcţia *size()*:

```
template <typename KEY_NODE_TYPE>
inline int
btree_node<KEY_NODE_TYPE>::size( void ) const
```

```
{ return this->node_size ; }
```

Pentru a se determina numărul maxim de chei care pot fi păstrate într-un nod se foloseşte funcţia *capacity()*:

```
template <typename KEY_NODE_TYPE>
inline int
btree_node<KEY_NODE_TYPE>::capacity( void ) const
{ return this->node_capacity ; }
```

Un nod poate fi interogat pentru starea în care se află folosind funcţia *status()*:

```
template <typename KEY_NODE_TYPE>
inline btree_node<KEY_NODE_TYPE>::BTREE_NODE_STATUS
btree_node<KEY_NODE_TYPE>::status( void ) const {
  if( 0 == size() ) return EMPTY ;
  else if( size() < ( capacity() / 2 ) ) return UNDERFLOW ;
  else if( size() == ( capacity() / 2 ) ) return MINIMAL ;
  else if( size() == capacity() ) return FULL ;
  else if( size() > capacity() ) return OVERFLOW ;
  return OPERATIONAL ;
}
```

Se observă că starea nodului este funcţie de numărul de chei aflate la un moment dat în nod. Mai exact, dacă avem un arbore B de ordin m, atunci numărul maxim de chei dintr-un nod va fi *m – 1*, iar numărul minim de chei va fi $\left\lceil \frac{m}{2} \right\rceil - 1$. Parametrul primit de constructor va fi *m – 1*, deci capacitatea nodului va fi *m – 1*. Cum numărul de chei aflate la un moment dat într-un nod se obţine folosind funcţia *size()*, vom avea:

- dacă *size()* returnează zero, atunci în nod nu se găseşte nici o cheie $\Rightarrow$starea nodului este *EMPTY* ;
- numărul minim de chei din nod este *capacity()/2*; deci dacă *size()==capacity()/2*, atunci nodul are numărul minim de chei $\Rightarrow$starea nodului este *MINIMAL* ;
- dacă *size() < capacity() / 2*, atunci nodul are prea puţine chei $\Rightarrow$starea nodului este *UNDERFLOW* ;
- dacă *size() == capacity()*, atunci nodul are numărul maxim de chei permise $\Rightarrow$starea nodului este *FULL* ;
- dacă *size() > capacity()*, atunci nodul are prea multe chei $\Rightarrow$starea nodului este *OVERFLOW*.

Pentru a se determina dacă nodul este un nod frunză se foloseşte funcţia *is_leaf()*, care va returna *true* dacă nodul este o frunză (un nod este nod frunză dacă nu are nici un descendent):

```
template <typename KEY_NODE_TYPE>
bool
btree_node<KEY_NODE_TYPE>::is_leaf( void ) const {
  assert( 0 != this->node_childs ) ;

  // return 0 == this->node_childs[0] ;
  for( int idx = 0; idx <= size(); idx++ )
    if( 0 != this->node_childs[idx] ) return false ;
```

```
  return true ;
}
```

Ca funcţii pentru interogarea valorii unei chei şi unui descendent dintr-o anume poziţie se folosesc funcţiile *key_value()* şi *child()*:

```
template <typename KEY_NODE_TYPE>
inline KEY_NODE_TYPE&
btree_node<KEY_NODE_TYPE>::key_value( int position ) {
  if( position < 0 ||
      position >= size()
  ) {
    /**     signal out of bounds*/
    assert( false ) ;
  }

  return this->node_keys[position] ;
}

template <typename KEY_NODE_TYPE>
inline btree_node<KEY_NODE_TYPE>*
btree_node<KEY_NODE_TYPE>::child( int position ) {
  if( position < 0 ||
      position > size()
  ) {
    /**     signal out of bounds          */
    assert( false ) ;
    return 0 ;
  }

  return this->node_childs[position] ;
}
```

Căutarea unei valori de cheie într-un nod se face folosind funcţia *find()*. Primul parametru primit de funcţie este valoarea de cheie care se caută în nod. După cum valoarea de cheie căutată se găseşte sau nu în nod, *find()* va returna *true* sau *false*, cu menţiunea ca al doilea parametru al funcţiei (*position*, care este un parametru de ieşire) va fi setat după cum urmează:

- daca valoarea de cheie căutată se găseşte în nod, atunci *position* este setat la indexul la care se găseşte cheia în nod;
- dacă valoarea de cheie căutată nu se găseşte în nod, *position* va indica indexul subarborelui în care s-ar putea găsi valoarea de cheie căutată.

```
template <typename KEY_NODE_TYPE>
bool
btree_node<KEY_NODE_TYPE>::find( const key_node_type& value,
int& position ) {
  bool ret_value ;
  position = -1 ;

  if( _less_than( value, this->node_keys[0] ) ) {
    position = 0 ;
    ret_value = false ;
  } else {
    for( position = size() - 1 ;
           _less_than( value, key_value( position ) ) &&
```

```
            position > 0;
            position--
    ) ;

    ret_value = _equal_to( value, this->node_keys[position] ) ;
    if( !ret_value ) position++ ;
  }

  return ret_value ;
}
```

Inserarea unei valori de cheie în nod se realizează folosind funcţia *push()*. În implementarea de faţă inserarea valorii de cheie într-un nod se face întotdeauna specificând şi pointerul către subarborele din dreapta valorii de cheie (subarbore care conţine toate valorile de cheie mai mari decât valoarea de cheie inserată). Dacă nodul este în starea *OVERFLOW* sau valoarea de cheie există deja în nod, funcţia va returna –1, fără a face nimic altceva. În urma determinării poziţiei pe care va fi inserată valoarea de cheie, ar putea fi necesară o deplasare către dreapta a valorilor de cheie din nod mai mari decât cea care se inserează (deplasarea se face împreună cu pointerii către descendenţi; funcţia folosită este *shift2right()*).

```
template <typename KEY_NODE_TYPE>
int
btree_node<KEY_NODE_TYPE>::push(
  const key_node_type&        value,
  btree_node_ptr                    child
) {
  if( OVERFLOW == status() ) return -1 ;

  if( EMPTY == status() ) {
    this->node_keys[0] = value ;
    this->node_childs[1] = child ;
    this->node_size = 1 ;
    return 0 ;
  }

  int key_position = -1 ;
  if( find( value, key_position ) ) {
    /**     duplicate key value           */
    return -1 ;
  }

  if( key_position < size() ) shift2right( key_position ) ;
  this->node_keys[key_position] = value ;
  this->node_childs[key_position + 1] = child ;
  this->node_size++ ;

  return 0 ;
}
```

Funcţia *shift2right()* este:

```
template <typename KEY_NODE_TYPE>
int
btree_node<KEY_NODE_TYPE>::shift2right( const int start ) {
  if( EMPTY == status() ||
      start <  0  ||
```

```
      start >= size()
    ) return -1 ;

  for( int idx = size(); idx > start; idx-- ) {
    this->node_keys[idx] = this->node_keys[idx - 1] ;
    this->node_childs[idx + 1] = this->node_childs[idx] ;
  }
  return 0 ;
}
```

Pentru eliminarea unei chei dintr-o anumită poziţie se va folosi funcţia *remove_at()* (practic, eliminarea presupune diminuarea cu unu a numărului de chei conţinute de nod şi o deplasare către stânga a valorilor de cheie şi a subarborilor aflaţi în dreapta poziţiei din care se şterge cheia).

```
template <typename KEY_NODE_TYPE>
int
btree_node<KEY_NODE_TYPE>::remove_at( const int position ) {
  if( -1 == shift2left( position ) ) return -1 ;
  this->node_size-- ;

  return 0 ;
}
```

Funcţia *shift2left()* este:

```
template <typename KEY_NODE_TYPE>
int
btree_node<KEY_NODE_TYPE>::shift2left( const int start ) {
  if( EMPTY == status() ||
      start < 0         ||
      start >= size()
    ) return -1 ;

  for( int idx = start + 1; idx < size(); idx++ ) {
    this->node_keys[idx - 1] = this->node_keys[idx] ;
    this->node_childs[idx] = this->node_childs[idx + 1] ;
  }

  return 0 ;
}
```

Atunci când starea unui nod este de *OVERFLOW*, acesta se va diviza. În urma divizării se va obţine un nod nou. Funcţia de divizare a unui nod este *split()*. Parametrul funcţiei *split()* este de ieşire, fiind de tipul *btree_node_tuple*:

```
typedef  struct {
  key_node_type   key_value ;
  btree_node_ptr   node_ptr ;
} btree_node_tuple ;
```

În urma procesului de divizare a unui nod va urca în nodul părinte cheia mediană şi un pointer către nodul nou format. Cheia mediană va fi câmpul *key_value* al structurii *btree_node_tuple*. Pointerul către nodul nou

format va fi câmpul *node_ptr* al structurii *btree_node_tuple*. Noul nod va conţine valorile de chei şi subarborii din dreapta cheii mediane.

```
template <typename KEY_NODE_TYPE>
int
btree_node<KEY_NODE_TYPE>::split(
  btree_node<KEY_NODE_TYPE>::btree_node_tuple*  tuple
) {
  if( 0 == tuple ) return 0 ;
  if( OVERFLOW != this->status() ) return 0;

  int median_position = this->size() / 2;
  tuple->key_value = this->key_value( median_position );

  btree_node_ptr new_node = new btree_node_type( this->capacity());
  if( 0 == new_node ) return -1;

  for( int idx = median_position + 1;
       idx < this->size() ;
       idx++
  ) new_node->push( this->key_value( idx ), this->child( idx + 1 ) );
  new_node->node_childs[0] = this->child( median_position + 1 );

  this->node_size = median_position;
  tuple->node_ptr = new_node;

  return 0 ;
}
```

În continuare, va fi dată declaraţia clasei de arbore B:

```
template <typename KEY_TYPE>
class btree {

public:
  typedef KEY_TYPE                          key_type ;
  typedef btree_node<KEY_TYPE>              btree_node_type ;
  typedef btree_node_type*                  btree_node_ptr ;
  typedef btree_node_type::btree_node_tuple btree_node_tuple ;
  typedef btree<KEY_TYPE>                   btree_type ;
  typedef btree_type*                       btree_ptr ;

public:
  btree( int order ) ;
  virtual ~btree( void ) ;

  int push( const key_type& value ) ;
  int remove( const key_type& value ) ;

protected:
  int push_down( btree_node_tuple* tuple, btree_node_ptr current ) ;

  int remove_down( const key_type& value, btree_node_ptr current ) ;
  int replace_with_predecessor( btree_node_ptr node, int position ) ;
  void restore( btree_node_ptr current, const int position ) ;
  void move_left( btree_node_ptr current, const int position ) ;
  void move_right( btree_node_ptr current, const int position ) ;
  void combine( btree_node_ptr current, const int position ) ;

private:
```

```
  btree_node_ptr  root ;
  int             order ;
} ; /**     class btree                                          */
```

Clasa *btree* este o clasă *template* după tipul valorii de cheie. La fel ca la clasa *btree_node* au fost folosite o serie de typedef-uri în cadrul clasei (codul este mai uşor de scris / citit dacă tipul *btree_node<KEY_TYPE>* se redefineşte ca fiind *btree_node_type*). Nodul rădăcină al arborelui B este dat de câmpul *root*. Ordinul arborelui B este dat de câmpul *order*.

Constructorul *btree<KEY_TYPE>::btree()* primeşte ca parametru ordinul arborelui şi setează rădăcina arborelui la 0.

```
template <typename KEY_TYPE>
btree<KEY_TYPE>::btree( int order ) {
  this->order = order ;
  this->root  = 0 ;
}
```

Destructorul clasei *btree<KEY_TYPE>::~btree()* va elibera spaţiul de memorie ocupat de arbore.

```
template <typename KEY_TYPE>
btree<KEY_TYPE>::~btree( void ) {
  clear( this->root ) ;
  this->root = 0 ;
}

template <typename KEY_TYPE>
void btree<KEY_TYPE>::clear( btree_node_ptr node ) {
      if( 0 == node ) return ;
      if( !node->is_leaf() )
            for( int idx = 0 ; idx <= node->size(); idx++ )
                  clear( node->child( idx ) ) ;
      delete node ;
}
```

Funcţia de inserare a unei valori de cheie în arbore este *push()*. Dacă arborele nu are nici o valoare de cheie inserată (adică rădăcina arborelui este 0 $\Leftrightarrow$ arborele este gol), atunci inserarea valorii de cheie este simplă: se construieşte rădăcina arborelui cu valoarea de cheie care se inserează. Daca arborele nu este gol, atunci se inserează cheia în arbore recursiv folosind funcţia *push_down()*, pornind de la nodul rădăcină. Funcţia *push_down()* va returna 1 dacă în urma procesului de inserare recursivă a valorii de cheie nodul rădăcină a fost divizat (cheia şi un pointer către subarborele drept al cheii vor fi conţinute de câmpurile variabilei *tuple*). În acest caz înălţimea arborelui creşte cu unu, formându-se o nouă rădăcină cu valoarea de cheie dată de câmpul *kei_value* al variabilei *tuple*; subarborele stâng va fi vechea rădăcină a arborelui, iar subarborele drept va fi dat de câmpul *node_ptr* al variabilei *tuple*.

```
template <typename KEY_TYPE>
int btree<KEY_TYPE>::push( const key_type& value ) {
  if( 0 == this->root ) {
    this->root = new btree_node_type( this->order - 1 ) ;
    if( 0 == this->root ) return -1 ;
```

```
    this->root->push( value, (btree_node_ptr)0 ) ;
    return 0 ;
  }

  btree_node_tuple tuple ;
  tuple.key_value = value ;
  tuple.node_ptr = 0 ;

  if( push_down( &tuple, this->root ) ) {
    btree_node_ptr new_root = new btree_node_type( this->order - 1 );
    if( 0 == new_root ) return -1 ;

    new_root->push( tuple.key_value, tuple.node_ptr ) ;
    new_root->node_childs[0] = this->root ;
    this->root = new_root ;
  }

  return 0 ;
}
```

Funcţia *push_down()* de inserare recursivă în arbore primeşte ca parametri nodul curent *current* în care se încearcă inserarea, şi, împachetat în variabila *tuple*, valoarea de cheie care se inserează. Iniţial, la primul apel, câmpul *node_ptr* al variabilei *tuple* va avea valoarea 0. În cadrul algoritmului de inserare se va căuta valoarea de cheie în nodul curent. Dacă aceasta există deja în nod (arbore), funcţia de inserare nu va face nimic şi va returna –1. Altfel, variabila *key_position* va indica:

- dacă nodul curent este frunză, *key_position* indică locul în care se va insera valoarea de cheie;
- dacă nodul curent nu este o frunză, *key_position* va indica subarborele în care va trebui să se facă inserarea.

Dacă nodul curent este frunză, inserarea este simplă: se apelează doar metoda *push()* de inserare a unei chei într-un nod (la inserarea într-un nod frunză întotdeauna câmpul *node_ptr* al parametrului *tuple* va fi 0). Dacă nodul curent nu este frunză, atunci va trebui urmărit dacă în urma inserării recursive în nodul curent a urcat o valoare de cheie. Daca în nodul curent a urcat o valoare de cheie (metoda *push_down()* a returnat valoarea 1) atunci:

- câmpurile parametrului *tuple* vor conţine valoarea de cheie care a urcat şi un pointer către subarborele drept corespunzător cheii ;
- în nodul curent vor trebui inserate *key_value* şi *node_ptr* indicate de câmpurile parametrului *tuple*.

În final, se verifică starea în care se află nodul curent. Dacă starea acestuia este de *OVERFLOW*, atunci înseamnă ca acesta conţine prea multe chei şi va trebui divizat. Pentru aceasta se foloseşte metoda *split()* a nodului care va primi ca parametru adresa lui *tuple*. În urma apelului metodei *split()* conţinutul câmpurilor *key_value* şi *node_ptr* ale variabilei *tuple* vor fi actualizate pentru a indica valoarea de cheie care va urca în nodul părinte al nodului curent şi pointerul către subarborele drept; în acest caz metoda *push_down()* va returna 1 pentru a indica faptul că în nodul părinte vor trebui folosite câmpurile variabile *tuple*. Dacă starea nodului nu este *OVERFLOW*, metoda *push_down()* va returna 0.

```
template <typename KEY_TYPE>
int btree<KEY_TYPE>::push_down(
```

```
  btree_node_tuple* tuple,
  btree_node_ptr    current
) {
  if( 0 == current ) return 0 ;

  int key_position ;
  bool duplicate_key = current->find( tuple->key_value, key_position)
;
  if( duplicate_key ) {
    /**     signal duplicate value                                    */
    return -1 ;
  }

  if( current->is_leaf() ) {
    current->push( tuple->key_value, tuple->node_ptr );
  } else {
    if( push_down( tuple, current->child( key_position ) ) )
      current->push( tuple->key_value, tuple->node_ptr);
  }

  if( btree_node_type::OVERFLOW == current->status() ) {
    current->split( tuple ) ;
    return 1 ;
  }

  return 0 ;
}
```

13.4 Algoritmii C++ pentru ştergerea unei valori de cheie intr-un arbore B

Funcţia de ştergere a unei valori de cheie dintr-un arbore B este *remove()*. Intern este folosită funcţia recursivă *remove_down()*. Dacă în urma ştergerii valorii de cheie nodul rădăcină nu mai are nici o valoare de cheie (starea nodului rădăcină este *EMPTY*), atunci noua rădăcină a arborelui este subarborele stâng al vechii rădăcini. Vechea rădăcină se elimină din memorie.

```
template <typename KEY_TYPE>
int btree<KEY_TYPE>::remove( const key_type& value ) {
  remove_down( value, this->root ) ;

  if( btree_node_type::EMPTY == this->root->status() ) {
    btree_node_ptr old_root = this->root ;
    this->root = this->root->child( 0 ) ;

    delete old_root ;
  }

  return 0 ;
}
```

Funcţia de ştergere recursivă a unei valori de cheie din arbore este *remove_down()*. Parametrii primiţi de funcţie sunt valoarea de cheie care se şterge şi un pointer către nodul curent. Dacă la un moment dat nodul curent este 0, atunci înseamnă că valoarea de cheie solicitată a fi ştearsă nu se

găseşte în arbore. Altfel, funcţie de tipul nodului unde cheia este găsită avem următoarele cazuri:

- dacă valoarea de cheie se găseşte într-un nod frunză, atunci ştergerea înseamnă apelarea metodei *remove()* a nodului frunză pentru eliminarea cheii aflată în poziţia dată de variabila *position* ;
- dacă valoarea de cheie a fost găsită într-un nod care nu este nod frunza, atunci valoarea de cheie care trebuie ştearsă va fi înlocuită, în implementarea de faţă, cu valoarea de cheie care o precede (se poate demonstra că valoarea de cheie care o precede se găseşte într-un nod frunză). După ce se face înlocuirea folosind funcţia *replace_with_predecessor()*, se va continua algoritmul de ştergere, dar de data aceasta se va solicita ştergerea valorii de cheie cu care s-a făcut înlocuirea.

Se observă că dacă nodul curent este valid (nu este 0) şi valoarea de cheie care se caută nu se găseşte în nodul curent, atunci variabila *position* va fi poziţionată de metoda *find()* a nodului ca fiind poziţia subarborelui care este posibil să conţină valoarea de cheie.

În finalul funcţiei, dacă nodul curent nu este frunză (este un nod părinte care are descendenţi care au fost afectaţi), se testează starea nodului rădăcină pentru subarborele pe care s-a efectuat coborârea în procesul de ştergere. Dacă starea acestui nod este *UNDERFLOW* (conţine prea puţine valori de cheie), atunci va fi apelată funcţia *restore()* care va restaura proprietăţile de arbore B.

```
template <typename KEY_TYPE>
int btree<KEY_TYPE>::remove_down(
  const key_type& value,
  btree_node_ptr current
) {
  if( 0 == current ) {
    /**     signal value not found   */
    return 0 ;
  }

  int position ;
  if( current->find( value, position ) ) {
    if( current->is_leaf() ) current->remove_at( position ) ;
    else {
      replace_with_predecessor( current, position ) ;
      remove_down(current->key_value(position ),
current->child( position)) ;
    }
  } else remove_down( value, current->child( position ) ) ;

  if( !current->is_leaf() &&
      btree_node_type::UNDERFLOW    =    =    current->child(position)-
>status() )
    restore( current, position ) ;

  return 0 ;
}
```

Funcţia de înlocuire a unei valori de cheie cu valoarea de cheie care o precede este *replace_with_succesor()*. Parametrii primiţi de funcţie sunt: *node*, un pointer către nodul care conţine valoarea de cheie care se înlocuieşte şi *position* care dă poziţia subarborelui (în nodul indicat de *node*)

care conţine valoarea de cheie care precede valoarea de cheie care se înlocuieşte.

```
template <typename KEY_TYPE>
int btree<KEY_TYPE>::replace_with_predecessor(
  btree_node_ptr  node,
  int             position
) {
  if( position > node->size() ) return 0 ;

  btree_node_ptr leaf = node->child( position ) ;
  while( !leaf->is_leaf() ) leaf = leaf->child( leaf->size() ) ;
  node->node_keys[position] = leaf->key_value( leaf->size() - 1 ) ;

  return 1 ;
}
```

Funcţia care asigură restaurarea proprietăţilor de arbore B este *restore()*. Parametrii funcţiei sunt: un pointer către un nod părinte (*current*) şi o poziţie (*position*) care indică un subarbore al nodului *current* (nodul rădăcină al acestui subarbore conţine prea puţine valori de cheie). Funcţia folosită în implementarea de faţă este cumva orientată către stânga, în sensul că mai întâi se uită la nodul vecin din stânga pentru a lua o valoare de cheie, folosind nodul vecin din dreapta numai dacă nu găseşte suficiente valori de cheie în nodul din stânga. Paşii care sunt necesari sunt ilustraţi în figura 13.22.

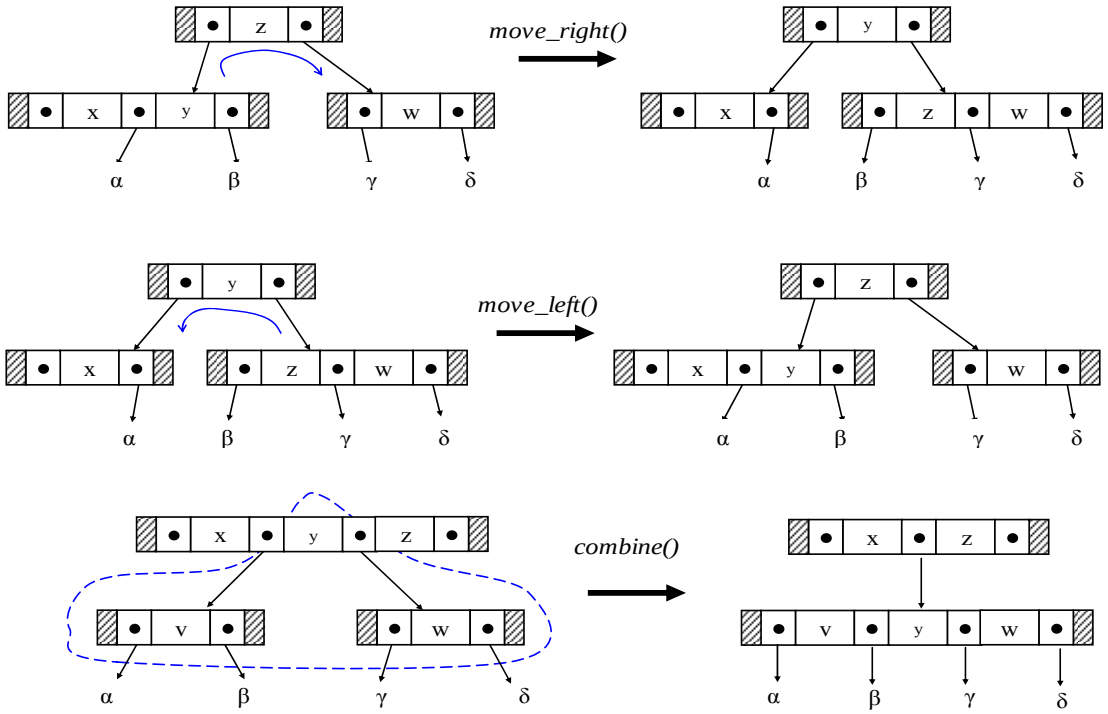

Figura 13.22 *Paşii parcurşi în funcţia de restaurare a proprietăţilor unor arbore B*

```
template <typename KEY_TYPE>
void btree<KEY_TYPE>::restore(
  btree_node_ptr  current,
```

```
  const int     position
) {
  if( 0 == position ) {
    if( current->child( 1 )->status() > btree_node_type::MINIMAL )
      move_left( current, 1 ) ;
    else
      combine( current, 1 ) ;
  } else if( position == current->size() ) {
    if(current->child(current->size()-1)->status()>
btree_node_type::MINIMAL)
      move_right( current, position ) ;
    else
      combine( current, position ) ;
  }   else   if(   current->child(   position   -   1   )->status()   >
btree_node_type::MINIMAL )
    move_right( current, position ) ;
  else   if(   current->child(   position   +   1   )->status()   >
btree_node_type::MINIMAL )
    move_left( current, position + 1 ) ;
  else
    combine( current, position );
}
```

Funcţiile *move_left()*, *move_right()* şi *combine()* sunt uşor de dedus din figura 13.22.

```
template <typename KEY_TYPE>
void btree<KEY_TYPE>::move_left(
  btree_node_ptr  current,
  const int   position
) {
  btree_node_ptr node = current->child( position - 1 ) ;
  node->push(
      current->key_value( position - 1 ),
      current->child( position )->child( 0 )
  ) ;

  node = current->child( position ) ;
  current->node_keys[position - 1] = node->key_value( 0 );
  node->node_childs[0] = node->node_childs[1] ;
  node->shift2left( 0 ) ;
  node->node_size-- ;
}

template <typename KEY_TYPE>
void btree<KEY_TYPE>::move_right(
  btree_node_ptr  current,
  const int   position
) {
  btree_node_ptr node = current->child( position ) ;
  node->shift2right( 0 ) ;
  node->node_childs[1] = node->child( 0 ) ;
  node->node_keys[0] = current->key_value( position - 1 ) ;
  node->node_size++ ;
  node = current->child( position - 1 ) ;
  current->node_keys[position - 1] = node->key_value(node->size()-1);
  current->child(position)->node_childs[0]=node->child(node->size());
  node->node_size-- ;
}
```

```
template <typename KEY_TYPE>
void btree<KEY_TYPE>::combine(
  btree_node_ptr  current,
  const int   position )
{
  btree_node_ptr rnode = current->child( position ) ;
  btree_node_ptr lnode = current->child( position - 1 ) ;

  lnode->push(current->key_value( position - 1 ), rnode->child( 0 ));
  for( int idx = 0; idx < rnode->size(); idx++ )
      lnode->push( rnode->key_value( idx ), rnode->child( idx + 1 ) );

  current->remove_at( position - 1 ) ;
  delete rnode ;
}
```