13 INDEXING AND MULTIWAY TREES

OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Indexing techniques
- B-trees which prove invaluable for problems of external information retrieval
- A class of trees called tries, which share some properties of table lookup
- Important uses of trees in many search techniques

An important area in computer science is information retrieval. An information retrieval application, a database, which may contain a wide variety of data structures, is maintained on an online basis using large random access files. These files are searched for requested information based on index items generated from a user query. One of the problems associated with information retrieval systems and especially automated library systems is creating a good indexing scheme. We shall learn about indexing schemes in this chapter.

13.1 INTRODUCTION

A *file* is a collection of records, each record having one or more fields. The fields used to distinguish among the records are known as *keys*. *File organization* describes the way in which records are stored in a file. File organization is concerned with representing data records on an external storage media. The choice of such a representation depends on the environment where the file is to operate, for example, real-time, batched, simple query, one key, or multiple keys. When there is only one key, the records may be stored on this key and stored sequentially either on a tape or a disk. This results in a sequentially ordered file. This organization is good for files operating in batched retrieval and update modes when the number of transactions batched is large enough to make the processing cost effective. When the number of keys is more than one or when real time responses are needed, a sequential organization is not adequate. In a general situation, several indices may have to be maintained. In these cases, the file organization breaks down into two more aspects:

Directory for the collection of indices

File organization for the physical organization of records

Many alternative file organizations exist, each suitable in a particular situation. File organization is the way records are organized on a physical storage. One such organization is *sequential* (ordered and unordered). In this general framework, processing a query or updating a request would proceed in two steps:

1. The indices would be interrogated to determine the parts of the physical file to be searched.
2. These parts of the physical file will be searched.

Depending upon the kinds of indices maintained, the second stage may involve only the accessing of records satisfying the query or may involve retrieving non-relevant records too.

Let us study about indexing and the different indexing schemes.

13.2 INDEXING

One of the most popular indices is a book index. An index of a book is a table containing a list of topics (keys) and page numbers where the topic can be found (reference fields).

An index, whether it is a book or a data file index (in computer memory), is based on the basic concepts such as keys and reference fields. The index to a book provides a way to find a topic quickly. Imagine a book that does not have a good index. Then, we have only one solution, that is, to scan the whole book sequentially for finding a particular topic. In general, indexing is a way of finding things quickly.

To search some topics in a book is a problem which cannot be solved by methods we have studied in Chapter 9, searching and sorting. Rearranging all the words in the book in alphabetical order certainly would make finding any particular term easier, but would obviously have disastrous effects on the meaning of the book. Even though this book example, where the words in the book are referred to as pinned records, is absurd, it clearly underscores the power and importance of the index as a conceptual tool. Indexing works on indirect addressing. An index lets us impose order on a file without rearranging the file.

One more example where indexing is used is a library. To locate a book by a specific author, title, or subject, we can take the card catalog. The card catalog is actually a set of three indices, each using a different key field and all of them using the same catalog number as a reference field. Another use of indexing is to provide multiple access paths to a file. The advantage of indexing is that it gives keyed access to variable length records.

13.2.1 Indexing Techniques

A directory is a collection of indices. It may contain one index for every key or only one index for some of the keys. If an index contains an entry for every record, then it is called a *dense index*. If an index contains an entry for only some of the records, then it is *non-dense index*. In some cases, all the indices may be integrated into one large index.

The index is a collection of pairs of the form (key value, address). For example, consider the sample data for employee file as in Table 13.1.

Table 13.1 Employee records

Record	Emp. no.	Name	Occupation	Disk address
A	100	Saurabh	Developer	P1
B	500	Abolee	Project head	P2
C	300	Anagha	Developer	P3
D	200	Abhijeet	Project head	P4
E	400	Devnarayanan	Developer	P5

Suppose P1, P2, P3, P4, P5 are the disk addresses where these records are stored. Let 'Emp. no.' be the key. Then, the index will have the entries (100, P1), (500, P2), (300, P3), (200, P4), and (400, P5). This is a dense index because the key is distinct for all records and there is an entry for each record. If we keep 'Occupation' as the key, then the index will be (Developer, q1), (Project Head), q2), where q1 is a disk address that stores the list of addresses of all developers, that is, P1, P3, and P5, and q2 is a disk address that stores the list of addresses of all project heads, that is, P2 and P4. This is also a dense index.

Index can also be maintained as the key value—address1, address2, …, address*n*. However, if the number of records associated with each key varies, then it results in variable size nodes and complex storage management.

Different operations on the index are searching a key, modifying some entry in the index, inserting a new entry, and deleting an entry from the index. An index is too large and has to be maintained on the external storage. Let us see some indexing techniques.

Cylinder-surface Indexing

This is the simplest type of index organization. It is useful only for the primary key index of a sequentially ordered file. In a sequentially ordered file, the physical sequence of records is ordered by the key, called the *primary key*. The employee file in Table 13.1 is not sequentially ordered if 'Emp. no.' is a primary key because that field is not sorted. The sequentially ordered file can be stored on a tape or a disk. Disk memory has many surfaces, each surface having tracks. A cylinder *j* consists of track *j* on all the surfaces. So, the sequential interpretation of disk memory can be done in the following way. First, all tracks on cylinder 1 are accessed, then cylinder 2, and so on. So the read/write heads are moved one cylinder at a time. This is shown in Fig. 13.1.

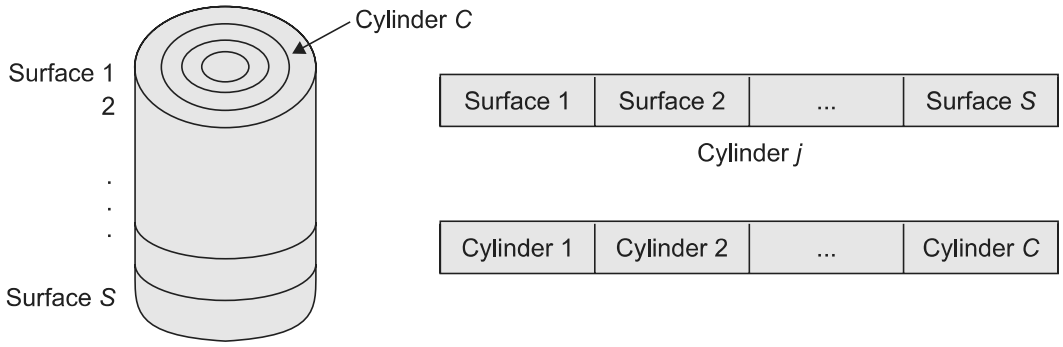


Fig. 13.1 Cylinder-surface indexing

The cylinder-surface index consists of a cylinder index and several surface indices. If the file requires 1 through C cylinders, then there are C entries in the cylinder index. There is one entry corresponding to the largest key value in each cylinder. For each cylinder, there is a surface index. If the disk has S usable surfaces, then each surface index has S entries. The total number of surface index entries is $C \times S$. For example, consider Table 13.2.

Table 13.2 Employee records cylinder-surface indexing

Emp. no.	Emp. name	Cylinder	Surface
1	Abolee	1	1
2	Anand	1	1
3	Amit	1	2
4	Amol	1	2
5	Rohit	2	1
6	Santosh	2	1
7	Saurabh	2	2
8	Shila	2	2

Let there be two surfaces and two records stored per track. The file is organized sequentially on the field 'Emp. name'. The corresponding cylinder index is given in Table 13.3.

Table 13.3 Cylinder index for Table 13.2

Cylinder	Highest key value
1	Amol
2	Shila

The surface index for cylinder 1 is the surface highest key value: Anand, Amol.
The surface index for cylinder 2 is the surface highest key value: Santosh, Shila.

A search for a record with a particular key value K is done in the following way. First, the key cylinder index is read into memory. In general, it has a few hundred entries, so it fits in one track. The cylinder index is searched to determine the required cylinder number, and then, for this cylinder, its surface index is read into memory and searched for the track. Then, this track is read in and searched for the key. For example, if we search for a record with the key 'Rohit', then the cylinder index tells that the record is either on cylinder 2 or not in the file. If the surface index of cylinder 2 is searched, then it shows that the record is either on surface 1 or not in the file. So in the second track t2, 1 is read and searched for. The desired record is found on this track. So the total number of disk accesses to get a record is three—one for the cylinder index, one for the surface index, and one for the track of records. If the track sizes are very large, then a sector index is maintained. If several disks are used to store a file, then a disk index is also maintained.

This method of maintaining a file and index is referred to as indexed sequential access method (ISAM). It is the simplest file organization for single key files but not useful for multiple key files.

Hashed Indexing

The operations related to hashed indices are the same as those for hash tables. This has been discussed in detail in Chapter 11.

13.3 TYPES OF SEARCH TREES

We have studied BSTs (binary search trees), AVL trees, optimal binary search trees, and heaps in Chapter 7. These were binary trees with outgoing degree two. For large data, these trees grow to a great height. To avoid these problems, we retain the properties of BSTs and increase the outgoing degree more than two. In a BST, the node maintains two links for its left and right child, whereas in a multiway search tree, each node can maintain more than two links for its more than two subtrees. Such search trees have vast applications such as dictionary, spell checks, and external file indices.

13.3.1 Multiway Search Tree

Binary search trees generalize directly to multiway search trees. A multiway search tree is a tree of order m, where each node has utmost m children. Here m is an integer. If $k \leq m$ is the number of children, then the node contains exactly $k - 1$ keys, which partition all the keys in the subtrees into k subsets. If some of these subsets are empty, then the corresponding children in the tree are empty. Figure 13.2 shows a 5-way search tree.

We always want to construct a multiway search tree that will minimize file accesses. So the height of the tree should be as small as possible, for example, B-tree and B+ tree.

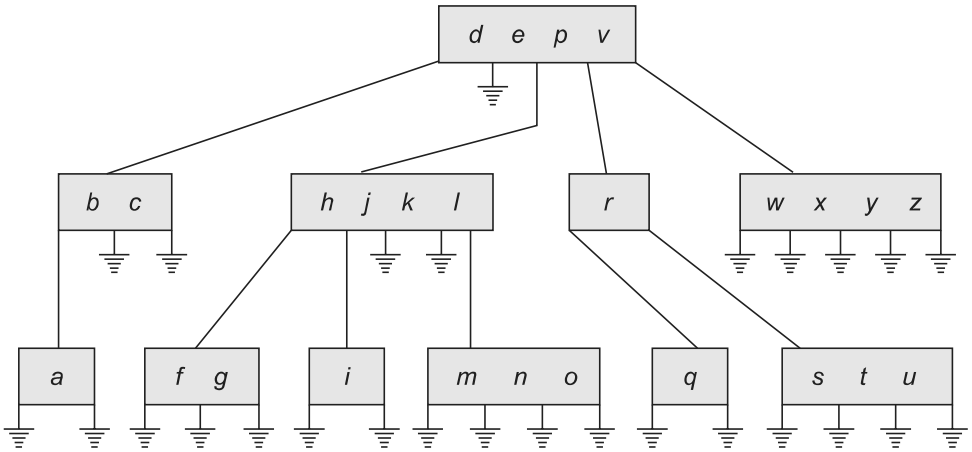


Fig. 13.2 5-way search tree

13.3.2 B-tree

When we want to locate and retrieve records stored in a disk file, the time required for a single access is thousand times greater for external retrieval than for internal information retrieval.

Our goal in external searching is to minimize the number of disk access since each access takes so long compared to internal computation. Multiway trees are especially appropriate for external searching.

A B-tree is a balanced multiway tree. A node of the tree contains many records or keys of records and pointers to children.

To reduce disk access, the following points are applicable:

1. Height is kept minimum.
2. All leaves are kept at the same level.
3. All nodes other than leaves must have at least minimum number of children.

B-tree Definition

A B-tree of order m is a multiway tree with the following properties:

1. The number of keys in each internal node is one less than the number of its non-empty children, and these keys partition the keys in the children in the fashion of the search tree.
2. All leaves are on the same level.
3. All internal nodes except the root have utmost m non-empty children and at least $\lceil m/2 \rceil$ non-empty children.
4. The root is either a leaf node, or it has from two to m children.
5. A leaf node contains no more than $m - 1$ keys.

Its node structure is given in Fig. 13.3.

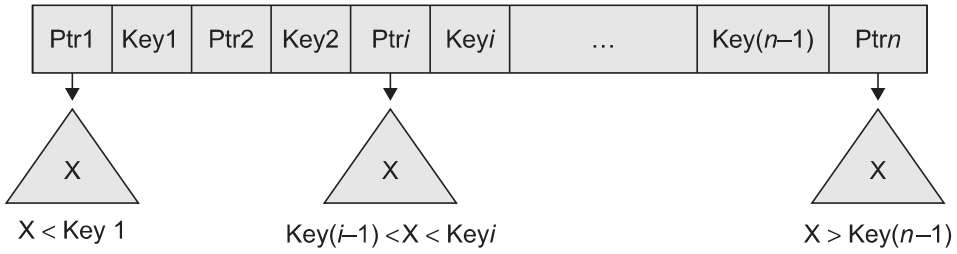


Fig. 13.3 Node structure for B-tree

The B-tree of order 5 for Fig. 13.3 shown in Fig. 13.4.

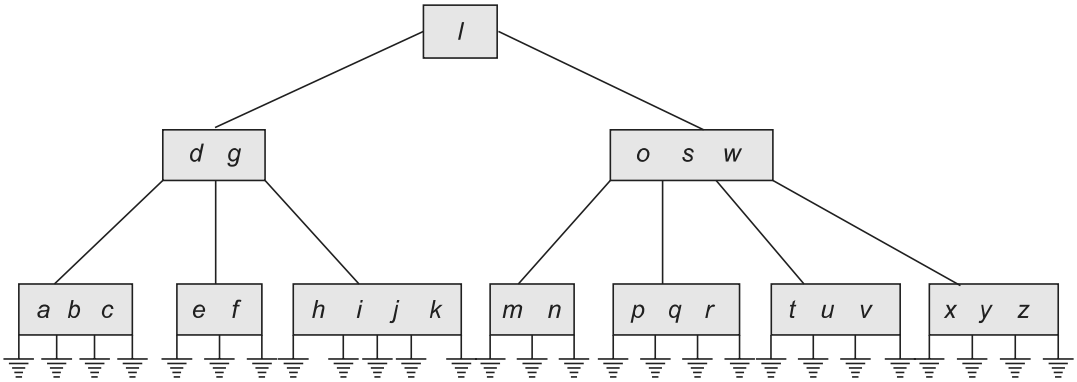


Fig. 13.4 B-tree of order 5

The maximum number of items in a B-tree of order m and height h is shown in Table 13.4.

Table 13.4 B-tree of order m and height h

Level	Number of keys
Root	$m - 1$
Level 1	$m(m - 1)$
Level 2	$m^2(m - 1)$
⋮	
Level h	$m^h(m - 1)$

So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) = [(m^{h+1} - 1)/ (m - 1)] (m - 1) = m^{h+1} - 1 \qquad (13.1)$$

When $m = 5$ and $h = 2$, Eq. (13.1) gives $5^3 - 1 = 124$.

We will describe a B-tree of order 5 using a C++ structure. The declaration of B-tree node is given in Fig. 13.5.

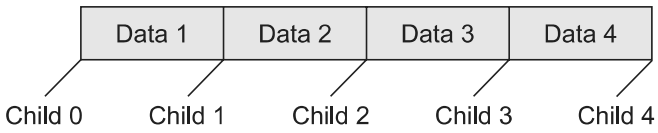


Fig. 13.5 Node structure of 5-way B-tree

Let us see how this can be implemented using the C++ code as in Program Code 13.1.

PROGRAM CODE 13.1

```
#define max 4
#define min 2
// Maximum number of keys in a node is m - 1,
// therefore max_keys = m - 1 = 5 - 1 = 4
// Minimum number of keys in a node is [m/2] - 1,
// therefore min_keys = [m/2] - 1
// = [5/2] - 1 = 2
class btnode
{
   public:
      int count;
      int data[max + 1];
      btnode *child[max + 1];
};

class btree
{
   int push_down(int, btnode*, int*, btnode**);
   void pushin(int, btnode*, btnode*, int);
   void split_node(int, btnode*, btnode*, int, int*,
   btnode**);
   void del_node(int, btnode*);
   void remove_key(btnode*, int);
   void successor(btnode*, int);
   void restore(btnode*, int);
   void move_right(btnode*, int);
   void move_left(btnode*, int);
   void combine_nodes(btnode*, int);
  int search_node(int, btnode*, int*);
```

```
    btnode*search(int, btnode*, int*);
    public:
       btnode* root;
       void display();
       btnode* del(int, btnode*);
       void pre_rec(btnode*);
       btnode* insert(int, btnode*);
};
```

Reasons for using B-trees B-trees are widely used for the following reasons:

1. The cost of each disk transfer is high when the searching tables are held on disk and do not depend much on the amount of data transferred, especially if the consecutive items are transferred. Consider a condition of the B-tree of order 101. We can transfer each node in one disk read operation.
2. A B-tree of order 101 and height 3 can hold $101^4 - 1$ items (approximately 100 million), and any item can be accessed with three disk reads (assuming we hold the root in memory).
3. When a balanced tree is required and if we take $m = 3$, we get a '2–3 tree', where the non-leaf nodes have two or three children (i.e., one or two keys).
4. B-trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree.

Operations on B-tree

The following are the operations performed on a B-tree.

Searching a node The function `search_node()` determines if the new key is in the current node and if not, finds which of the children should be searched for. This is described in Program Code 13.2.

PROGRAM CODE 13.2

```
/* Search_node() searches a new key in the current node.
If found returns its position in the current node, else
returns child which should be searched next */
int btree :: search_node(int newkey, btnode *curr, int
*pos)
{
   if(newkey < curr->data[1])
   {
      *pos = 0;
      return 0;
   }
```

```
    else
    {
       *pos = curr->count;
       while((newkey < curr->data[*pos]) && (*pos > 1))
          (*pos)--;
       if(newkey == curr->data[*pos])
          return 1;
       else
          return 0;
    }
}
```

Searching a B-tree In Program Code 13.3, the `search()` function traverses the B-tree.

PROGRAM CODE 13.3

```
btnode * btree :: search(int newkey, btnode *root, int
*pos)
{
   if(!root)
   {
      return null;
   }
   else if(search_node(newkey, root, pos))
      return root;
   else
      return search(newkey, root->child[*pos], pos);
}
```

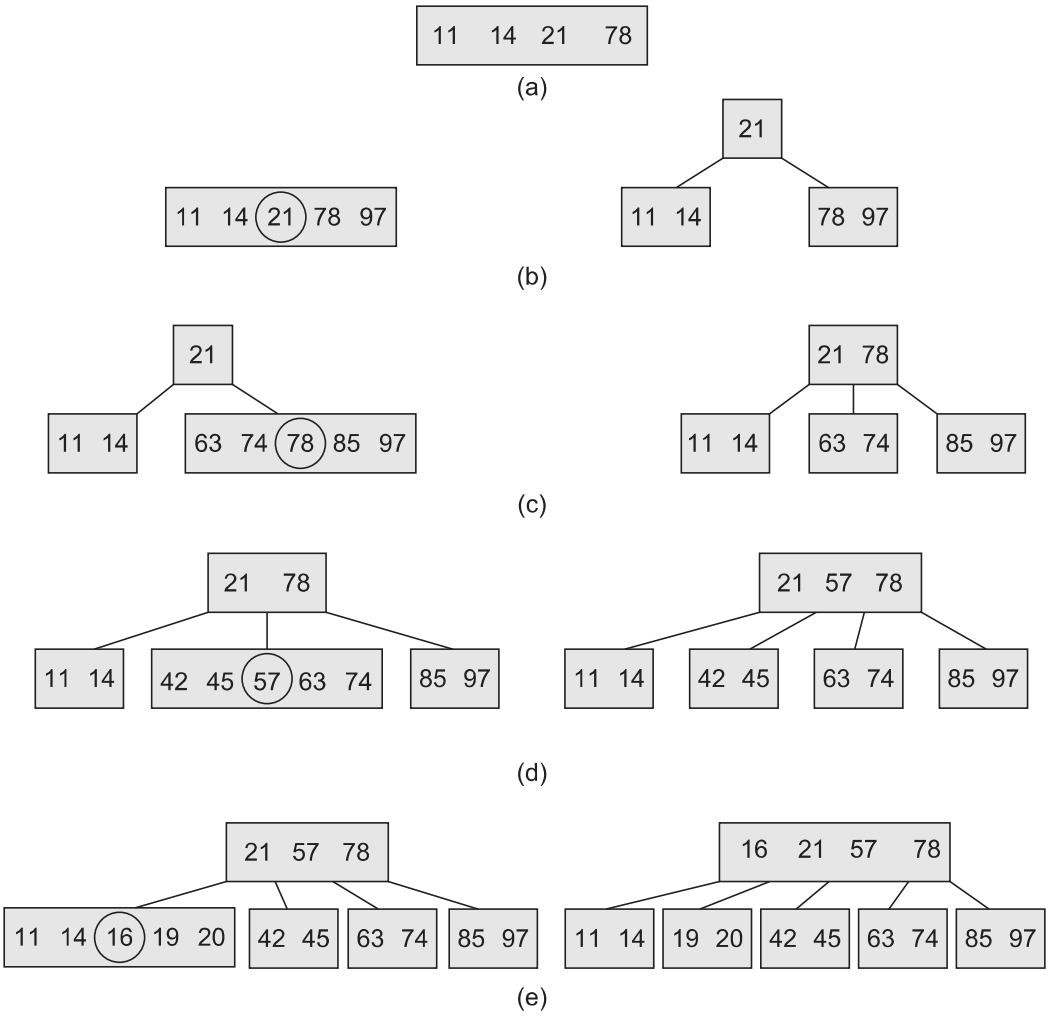
If a new key is present in it, then it returns the pointer to the node and the position of the new key in it; otherwise, it returns `null`.

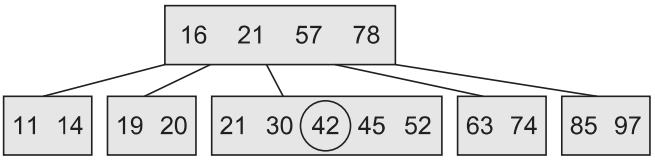
Inserting a key into a B-tree Binary search trees grow at their leaves, but the B-trees grow at the root. The general method of insertion is as follows:

1. First, the new key is searched in the tree. If the new key is not found, then the search terminates at a leaf.
2. Attempt to insert the new key into a leaf.
3. If the leaf node is not full, then the new key is added to it and the insertion is finished.
4. If the leaf node is full, then it splits into two nodes on the same level, except that the median key is sent up the tree to be inserted into the parent node.

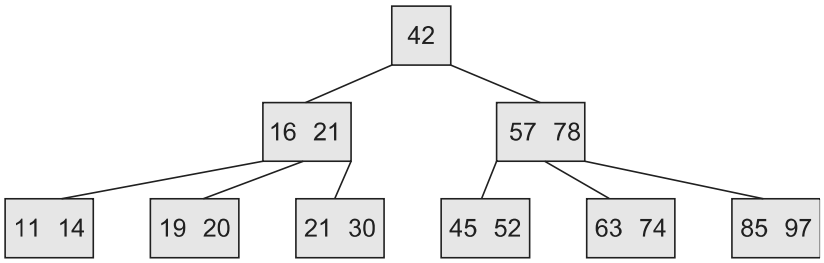
5. If this would result in the parent becoming too big, split the parent into two, promoting the middle key.
6. This strategy might have to be repeated all the way to the top.
7. If necessary, the root is split into two and the middle key is promoted to a new root, making the tree one level higher.

Let us see one example to build a B-tree of order 5 for the following data: 78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 52, 30, 21. This is illustrated in Figs 13.6(a)–(g). First the numbers 78, 21, 14, and 11 are inserted. The tree looks as in Fig. 13.6(a) post insertion. Then 97 is inserted, an overflow occurs at 21, and the tree is split as in Fig. 13.6(b). The numbers 85, 74, and 63 are inserted and again the tree is split as shown in Fig. 13.6(c). Figure 13.6(d) shows the split tree after insertion of 45, 42, and 57; Fig. 13.6(e) shows the split tree after insertion of 20, 16, and 19. Finally 52, 30, and 21 are inserted as shown in Fig. 13.6(f) and the final tree after split is shown in Fig. 13.6(g). The overflow in each step is depicted by encircling the number.





(f)



(g)

Fig. 13.6 Building a B-tree (a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4 (e) Step 5 (f) Step 6 (g) Final tree

In step 6, because of the overflow, data 42 moves up the root, and then the root becomes

So the root also overflows and splits. So 42 becomes the root of the final B-tree.
We should note two important points in the growth of B-trees.

1. When a node splits, it produces two nodes that are now only half full. So, later insertions may be made without any split again. Hence, one split prepares the way for several simple insertions.
2. It is always the median key that is sent upward. This improves the balance of the tree, no matter in what order the keys happen to arrive.

As shown in Fig. 13.7, the current node is split if it is full. After split, 'current' will be a left child and medright will be a right child, and meddata is a median key.

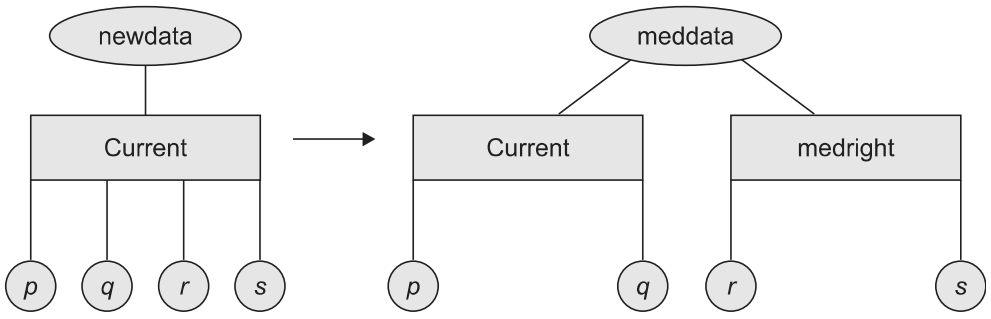


Fig. 13.7 Splitting the B-tree

The function `insert()` inserts `newdata` into the B-tree and then returns the root. This is shown in Program Code 13.4.

PROGRAM CODE 13.4

```
/* Function to insert newdata in B-tree */
btnode *btree :: insert(int newdata, btnode *root)
{
   int meddata;
   btnode *medright, *newroot;
   if(push_down(newdata, root, &meddata, &medright))
   {
      /* Tree if growing */
      newroot = new btnode;
      newroot->count = 1;
      newroot->data[1] = meddata;
      newroot->child[0] = root;
      newroot->child[1] = medright;
      return newroot;
   }
   return root;
}
```

In Program Code 13.5, `push_down()` recursively moves down the B-tree searching for new data. `newdata` is inserted into the subtree to which the node 'current' points. If `true` is returned, then the height of the subtree is increased and `meddata` should be reinserted higher in the tree, with subtree `medright` on its right.

PROGRAM CODE 13.5

```
Int btree :: push_down(int newdata, btnode *curr, int
*meddata, btnode **medright)
{
   int pos;
   if(curr == null)
   {
      /* cannot insert into empty subtree, so terminate */
      *meddata = newdata;
      *medright = null;
      return 1;
   }
  else
   {
      /* Search the current node */
     if(search_node(newdata, curr, &pos))
```

```
            cout << "\n\nError Duplicate Keys Cannot Be
            Inserted!!";
         if(push_down(newdata, curr->child[pos], meddata,
         medright))
         {
            if(curr->count < max)
            {
               /* Reinsert median key */
               pushin(*meddata, *medright, curr, pos);
               return 0;
            }
            else
            {
               /* Split node */
               split_node(*meddata, *medright, curr, pos,
               meddata, medright);
               return 1;
            }
         }
         return 0;
      }
   }
```

In Program Code 13.6, `pushin()` inserts the key `meddata` and its right-hand pointer `medright` into the node `*curr` at index `pos`.

PROGRAM CODE 13.6

```
void btree :: pushin(int meddata, btnode *medright,
btnode *curr, int pos)
{
   int p;
   for(p = curr->count; p > pos; p--)
   {
      /* Shift all the keys and child pointers to the
      right */
      curr->data[p + 1] = curr->data[p];
      curr->child[p + 1] = curr->child[p];
   }
   curr->data[pos + 1] = meddata;
   curr->child[pos + 1] = medright;
   curr->count++;
}
```

Splitting a full node The `split_node()` function splits a full node `*curr` with data `meddata`, and child pointer `medright` at index `pos` into nodes `*curr` and `*newright` and leaves the median key in the new median. The C++ code for splitting node is provided in Program Code 13.7.

PROGRAM CODE 13.7

```
void btree :: split_node(int meddata, btnode *medright,
btnode *curr, int pos, int *newmedian, btnode **newright)
{
   int p, median;
   if(pos <= min)
      median = min;
   else
      median = min + 1;
   /* Create a new node and put it on the right */
   *newright = new btnode;
   for(p = median + 1; p <= max; p++)
   {
      /* Move half the keys */
      (*newright)->data[p - median] = curr->data[p];
      (*newright)->child[p - median] = curr->child[p];
   }
   (*newright)->count = max - median;
   curr->count = median;
   if(pos <= min)
   {
      pushin(meddata, medright, curr, pos);
   }
   else
   {
     pushin(meddata, medright, *newright, pos - median);
   }
   *newmedian = curr->data[curr->count];
   (*newright)->child[0] = curr->child[curr->count];
   curr->count--;
}
```

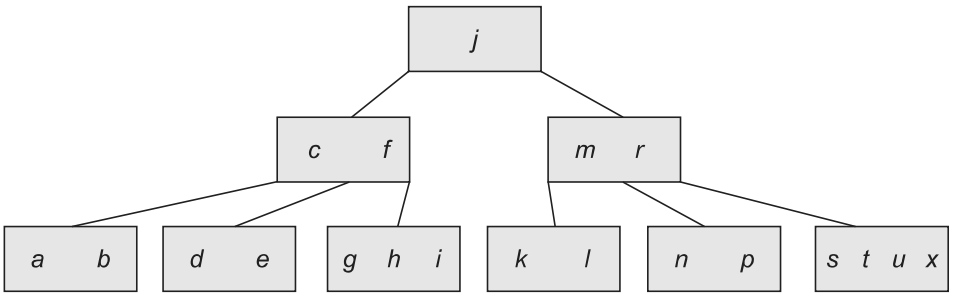
Deleting from a B-tree During insertion, the key always goes *into* a leaf. For deletion, if we wish to remove *from* a leaf, there are three possible ways mentioned as follows:

1. If the key is already in a leaf node and removing it does not cause that leaf node to have too few keys, then simply remove the key to be deleted.

2. If the key is *not* in a leaf, then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf—in this case, we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.
3. If these two conditions lead to a leaf node containing less than the minimum number of keys, then we have to look at the siblings immediately adjacent to the leaf in questions listed as follows:
   (a) If one of them has more than the minimum number of keys, then we can promote one of its keys to the parent and take the parent key into our lacking leaf.
   (b) If neither of them has more than the minimum number of keys, then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key), and the new leaf will have the correct number of keys; if this step leaves the parent with very few keys, then we repeat the process up to the root itself, if required.

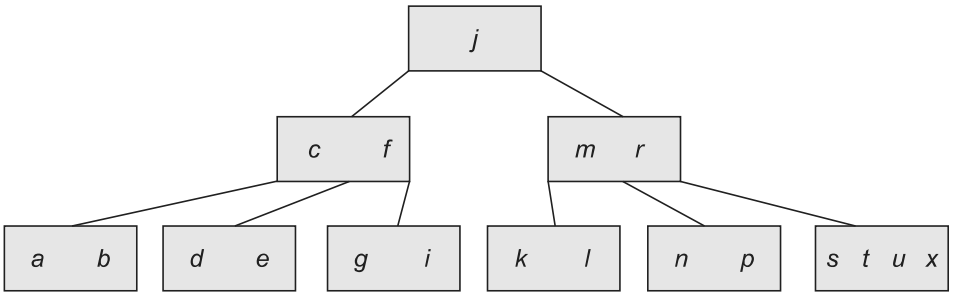
If the leaf contains more than the minimum number of entries, then the data can be deleted with no further action.

Consider the example as in Figs 13.8(a) and (b).



(a)

Now, delete *h*.



(b)

Fig. 13.8 Sample tree (a) Before deleting *h* (b) After deleting *h*

If the node contains the minimum number of entries, then look at the two leaves that are immediately adjacent to each other and are children of the same node. If one of these has

more than the minimum number of entries, then one of them can be moved into the parent node, and the entry from the parent can be moved into the leaf where the deletion occurs.

Figure 13.9 shows the B-tree when the leaf node *r* is deleted from Fig. 13.8(b).

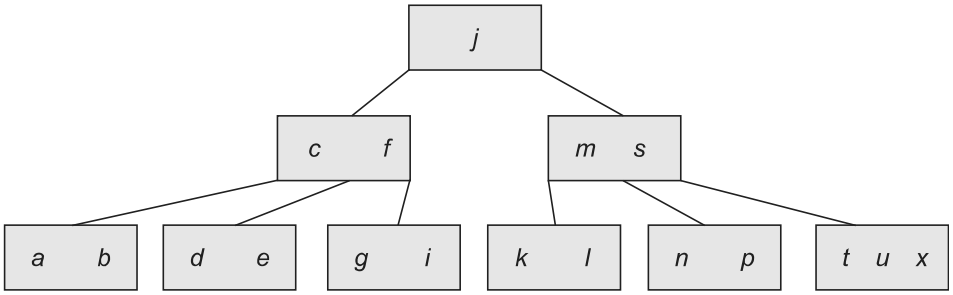


Fig. 13.9 Tree after *r* is deleted and *s* is moved to parent

Figure 13.10 shows the tree after deletion of *p*.

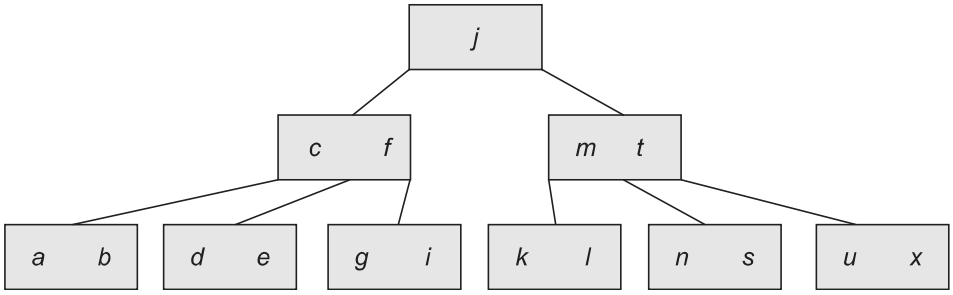
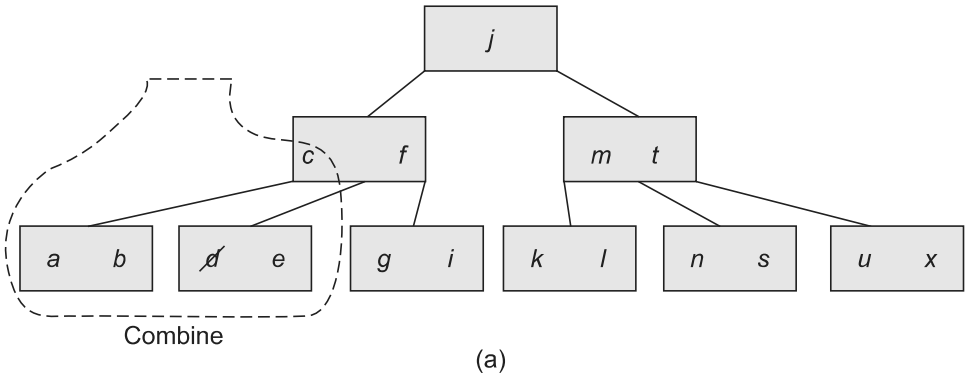


Fig. 13.10 Tree after *p* is deleted, *s* is moved down, and *t* is moved up to the parent

If the adjacent leaf has only the minimum number of entries, then the two leaves and the median entry from the parent can be combined as one new leaf, which will contain no more than the maximum number of entries allowed.

The process is repeated if required.

From the B-tree in Fig. 13.10, the leaf node *d* is deleted. The process of deleting and combining is shown in Figs 13.11(a)–(c).



(a)

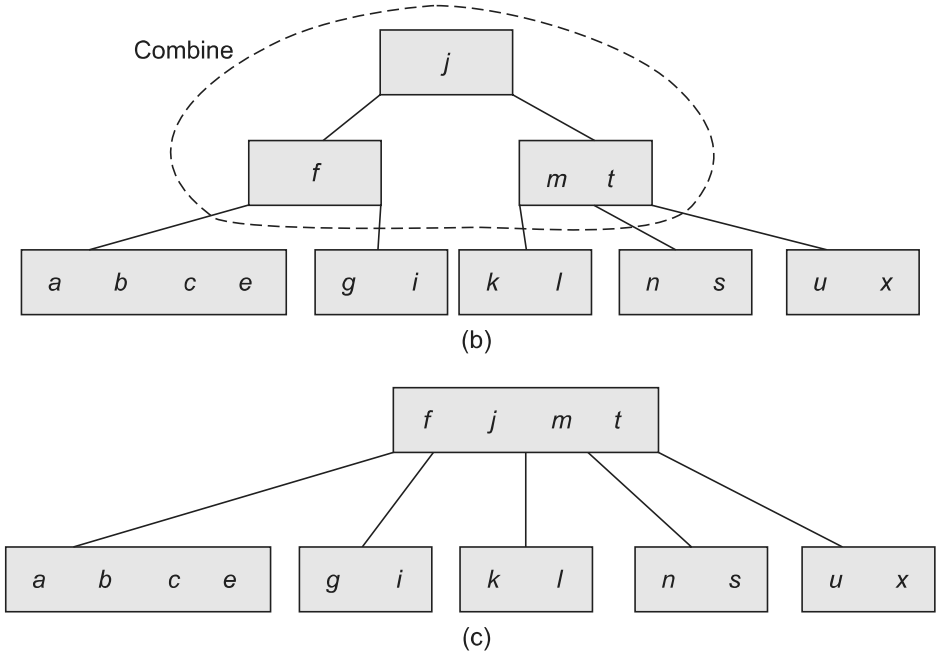


Fig. 13.11 Deleting and combining operations (a) Deletion of node *d* (b) Combining (c) Final B-tree

This is the final B-tree after the deletion of *d*. The combine process is repeated twice.

Let us see how the deletion operation can be implemented using C++ as shown in Program Code 13.8.

PROGRAM CODE 13.8

```
// DeleteBtree() deletes targetkey from the B-tree and
// returns the root
btnode *btree :: del(int key, btnode *root)
{
   btnode *oldroot;
   del_node(key, root);
   if(root->count == 0)
   {
      oldroot = root;
      root = root->child[0];
      delete oldroot;
   }
   return root;
}
```

In Program Code 13.9, `del_node()` searches the `targetkey` in the `curr` node. If it is found and the node is a leaf, then the immediate successor of the key is found and

is placed in the current node, and the successor is deleted. After deletion, the function checks to see if enough entries remain in the appropriate node, and if not, move entries as required.

PROGRAM CODE 13.9

```
void btree :: del_node(int key, btnode *curr)
{
   int pos;
   if(!curr)
   {
      cout << "\n\n Target Not Found";
      return ;
   }
   else
   {
      if(search_node(key, curr, &pos))
      {
         if(curr->child[pos - 1])
         {
            /* targetkey found, replace data [pos] by
            it successor */
            successor(curr, pos);
            del_node(curr->data[pos], curr->child[pos]);
         }
         else
            remove_key(curr,pos); /* removes key from
            pos of *current */
      }
      else      /* Target key not found in the current
                node, search a subtree */
         del_node(key, curr->child[pos]);
      if(curr->child[pos])
      {
          if(curr->child[pos]->count < min)
             restore(curr, pos);
      }
   }
}
```

The `remove_key()` function removes the target key from `pos` in the `curr` node and shifts the remaining keys one position ahead. The implementation of this operation is as in Program Code 13.10.

PROGRAM CODE 13.10

```
void btree :: remove_key (btnode *curr, int pos)
{
   int p;
   for(p = pos + 1; p <= curr->count; p++)
   {
      curr->data[p - 1] = curr->data[p];
      curr->child[p - 1] = curr->child[p];
   }
   curr->count--;
}

void btree :: successor (btnode *curr, int pos)
{
   btnode *leaf;
   leaf = curr->child[pos];
   while(leaf->child[0])
      leaf = leaf->child[0];
   curr->data[pos] = leaf->data[1];
}
```

The function `restore()` restores the minimum number of entries. It first searches the sibling on the left to take an entry and uses the right sibling only when there are no entries to spare in the left one. The working is shown in Program Code 13.11 using the function `restore()`.

PROGRAM CODE 13.11

```
void btree :: restore (btnode *curr, int pos)
{
   if(pos == 0)       /* leftmost key */
   {
      if(curr->child[1]->count > min)
         move_left(curr, 1);
      else
         combine_nodes(curr, 1);
   }
   else if(pos == curr->count)
   {
      if(curr->child[pos - 1]->count > min)
         move_right(curr, pos);
      else
         combine_nodes(curr, pos);
```

```
            /* Remaining cases */
      }
      else if(curr->child[pos - 1]->count > min)
         move_right(curr, pos);
      else if(curr->child[pos + 1]->count > min)
         move_left(curr, pos + 1);
      else
         combine_nodes(curr, pos);
}
```

Figure 13.12 shows the working of the `move_right()` function.

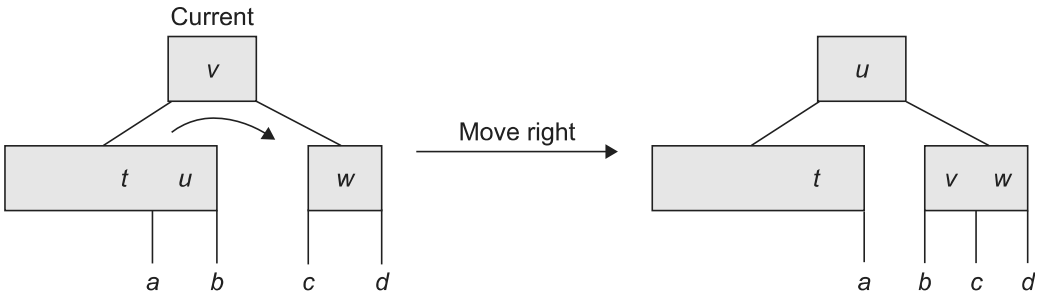


Fig. 13.12 Move right function

The `move_right()` function as given in Program Code 13.12, moves data from `*curr` node into the `child[pos]` and then moves the rightmost data from `child[pos - 1]` into the current node.

PROGRAM CODE 13.12

```
void btree :: move_right (btnode *curr, int pos)
{
   int p;
   btnode *temp;
   /* Set temp to right node of current */
   Temp = curr->child[pos];
   for(p = temp->count; p > 0; p--)
   {
      /* Shift all keys in the right node one position
      ahead */
      temp->data[p + 1] = temp->data[p];
      temp->child[p + 1] = temp->child[p];
   }
```

```
    temp->child[1] = temp->child[0];
    /* Increase count of right node */
    temp->count++;
    /* Move data from current to first place of right
    node */
    temp->data[1] = curr->data[pos];
    temp = curr->child[pos - 1];
    /* Move last data from left node into current */
    curr->data[pos] = temp->data[temp->count];
    curr->child[pos]->child[0] = temp->child[temp->count];
    /* Decrease count of left node */
    temp->count--;
}
```

Similarly, we can write a `move_left()` function given in Program Code 13.13, which moves data from `*curr` node into the `child[pos - 1]` and then moves the leftmost entry from `child[pos]` into `*curr` node.

PROGRAM CODE 13.13

```
void btree :: move_left (btnode *curr, int pos)
{
    int p;
    btnode *temp;
    temp = curr->child[pos - 1];
    /* Increase count of right node */
    /* Move data from current into last place of left
    node and increase its count */
    temp->count++;
    temp->data[temp->count] = curr->data[pos];
    temp->child[temp->count] = curr->child[pos]->child[0];
    /* Set temp to right node of current */
    temp = current→child[pos];
    /* Move data from first place of right node
    into last place of current and decrease count of
    right node */
   curr->data[pos] = temp->data[1];
    temp->child[0] = temp->child[1];
    temp->count--;
    /* Shift all keys in right node one position left */
   for(p = 1; p <= temp->count; p++)
```

```
    {
       temp->data[p] = temp->data[p + 1];
       temp->child[p] = temp->child[p + 1];
    }
}
```

Figure 13.13 illustrates the combining of nodes.

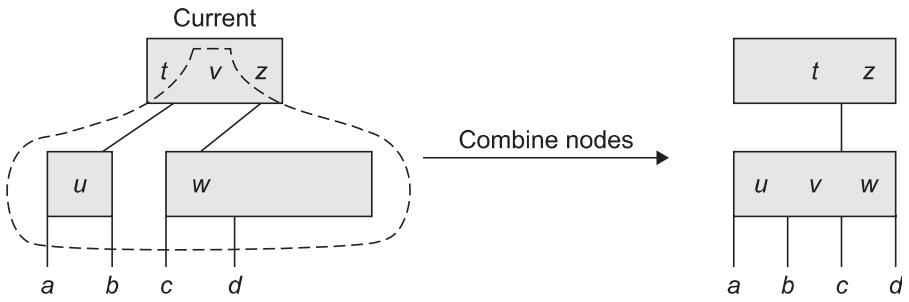


Fig. 13.13 Combining nodes

Program Code 13.14 elaborates the working of the function `combine_nodes()`.

PROGRAM CODE 13.14

```
void btree :: combine_nodes(btnode *curr, int pos)
{
   int p;
   btnode*left, *right;
   left = curr->child[pos - 1];
   right = curr->child[pos];
   /* Move data from current into left node and
   increase count of left, decrease count of current
   */
   left->count++;
   left->data[left->count] = curr->data[pos];
   left->child[left->count] = right->child[0];
   /* Copy all data and child pointers from right node
   into left node */
   for(p = 1; p <= right->count; p++)
  {
      left->count++;
      left->data[left->count] = right->data[p];
      left->child[left->count] = right->child[p];
   }
```

```
   /* Delete the data from current which is moved into
left node and shift the remaining data one position
left */
   for(p = pos; p < curr->count; p++)
   {
      curr->data[p] = curr->data[p + 1];
      curr->child[p] = curr->child[p + 1];
   }
   curr->count--;
   delete right;
}
```

This function combines the adjacent nodes at `child[pos - 1]` and `child[pos]` of `*curr` node into one node. In addition, data at `pos` in `*curr` node is moved into the combined node.

B-tree as Abstract Data Type

We have studied the implementation of various functions for a B-tree. The B-tree as an ADT is defined in Program Code 13.15.

PROGRAM CODE 13.15

```
/*****Implementation of B-tree*****/
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<process.h>
#define max 4
#define min 2
class btnode;
class qnode
{
   public:
      btnode* data;
      qnode* next;
      qnode(btnode* t){data = t; next = null;}
};

class queue
{
   qnode* front;
   qnode* rear;
```

```
   public:
      queue(){front = null; rear = null;}
      void add(btnode* t);
      btnode* remove();
      int isempty(){if(front == null)return 1;else
      return 0;}
};

void queue :: add(btnode* t)
{
   if(front == null)
      front = rear = new qnode(t);
   else
      rear = rear->next = new qnode(t);
}

btnode* queue :: remove()
{
   btnode* t;
   if(isempty())
      return 0;
   qnode* x = front;
   t = front->data;
   front = x->next;
   delete x;
   return t;
}

class btnode
{
   public:
      int count;
      int data[max + 1];
      btnode *child[max + 1];
};

class btree
{
   int push_down(int, btnode*, int*, btnode**);
   void pushin(int, btnode*, btnode*, int);
   void split_node(int, btnode*, btnode*, int, int*,
   btnode**);
   void del_node(int, btnode*);
```

```
    void remove_key(btnode*, int);
    void successor(btnode*, int);
    void restore(btnode*, int);
    void move_right(btnode*, int);
    void move_left(btnode*, int);
    void combine_nodes(btnode*, int);
    int search_node(int, btnode*, int*);
    btnode*search(int, btnode*, int*);
    public:
       btnode* root;
       void display();
       btnode* del(int, btnode*);
       void pre_rec(btnode*);
       btnode* insert(int, btnode*);
};

void btree :: display()
{
    queue q;
    btnode* m;
    m = root;
    while(m)
    {
       for(int i = 0; i < m->count; i++)
          cout << m->data[i] << "   ";
       for(i = 0; i < 5; i++)
       {
          if(m->child[i])
             q.add(m->child[i]);
       }
       m = q.remove();
       cout << "\n";
    }
}

void btree :: pre_rec(btnode *n)
{
    int i;
    if(n != null)
    {
       cout << endl << endl;
       for(i = 1; i <= n->count; i++)
```

```
        {
          cout << "\t" << n->data[i];
       }
       for(i = 0; i < n->count; i = i + 2)
       {
          pre_rec(n->child[i]);
          pre_rec(n->child[i + 1]);
       }
       if(n->count%2 == 0)
       {
          pre_rec(n->child[n->count]);
       }
    }
}

btnode * btree :: del(int key, btnode *root)
{
   btnode *oldroot;
   del_node(key, root);
   if(root->count == 0)
   {
      oldroot = root;
      root = root->child[0];
      delete oldroot;
   }
   return root;
}

void btree :: del_node(int key, btnode *curr)
{
   int pos;
   if(!curr)
   {
      cout << "\n\n Target Not Found";
      return;
   }
   else
   {
      if(search_node(key, curr, &pos))
         if(curr->child[pos - 1])
         {
            successor(curr, pos);
```

```
            del_node(curr->data[pos], curr->child[pos]);
         }
         else
         {
            remove_key(curr, pos);
         }
         else
            del_node(key, curr->child[pos]);
         if(curr->child[pos])
         {
            if(curr->child[pos]->count < min)
               restore(curr, pos);
         }
   }
}

void btree :: remove_key (btnode *curr, int pos)
{
   int p;
   for(p = pos + 1; p <= curr->count; p++)
   {
      curr->data[p - 1] = curr->data[p];
      curr->child[p - 1] = curr->child[p];
   }
   curr->count--;
}

void btree :: successor (btnode *curr, int pos)
{
   btnode *leaf;
   leaf = curr->child[pos];
   while(leaf->child[0])
      leaf = leaf->child[0];
      curr->data[pos] = leaf->data[1];
}

void btree :: restore (btnode *curr, int pos)
{
   if(pos == 0)
      if(curr->child[1]->count > min)
         move_left(curr, 1);
      else
```

```
            combine_nodes(curr, 1);
         else if(pos == curr->count)
            if(curr->child[pos - 1]->count > min)
            {
               move_right(curr, pos);
            }
            else
            {
               combine_nodes(curr, pos);
            }
            else if(curr->child[pos - 1]->count > min)
               move_right(curr, pos);
            else if(curr->child[pos + 1]->count > min)
               move_left(curr, pos + 1);
            else
               combine_nodes(curr, pos);
}

void btree :: move_right (btnode *curr, int pos)
{
   int p;
   btnode *temp;
   temp = curr->child[pos];
   for(p = temp->count; p > 0; p--)
   {
      temp->data[p + 1] = temp->data[p];
      temp->child[p + 1] = temp->child[p];
   }
   temp->child[1] = temp->child[0];
   temp->count++;
   temp->data[1] = curr->data[pos];
   temp = curr->child[pos - 1];
   curr->data[pos] = temp->data[temp->count];
   curr->child[pos]->child[0] = temp->child[temp-
   >count];
   temp->count--;
}

void btree :: move_left (btnode *curr, int pos)
{
   int p;
   btnode *temp;
```

```
    temp = curr->child[pos – 1];
    temp->count++;
    temp->data[temp->count] = curr->data[pos];
    temp->child[temp->count] = curr->child[pos]-
    >child[0];
    temp = curr->child[pos];
    curr->data[pos] = temp->data[1];
    temp->child[0] = temp->child[1];
    temp->count--;
    for(p = 1; p <= temp->count; p++)
    {
       temp->data[p] = temp->data[p + 1];
       temp->child[p] = temp->child[p + 1];
    }
}

void btree :: combine_nodes (btnode *curr, int pos)
{
    int p;
    btnode*left, *right;
    left = curr->child[pos – 1];
    right = curr->child[pos];
    left->count++;
    left->data[left->count] = curr->data[pos];
    left->child[left->count] = right->child[0];
    for(p = 1; p <= right->count; p++)
    {
       left->count++;
       left->data[left->count] = right->data[p];
       left->child[left->count] = right->child[p];
    }
    for(p = pos; p < curr->count; p++)
    {
       curr->data[p] = curr->data[p + 1];
       curr->child[p] = curr->child[p + 1];
    }
    curr->count--;
    delete right;
}

int btree :: search_node(int newkey, btnode *curr,
int *pos)
```

```
{
   if(newkey < curr->data[1])
   {
      *pos = 0;
      return 0;
   }
   else
   {
      *pos = curr->count;
      while((newkey < curr->data[*pos]) && (*pos > 1))
         (*pos)--;
      if(newkey == curr->data[*pos])
         return 1;
      else
         return 0;
   }
}

btnode * btree :: search(int newkey, btnode *root, int
*pos)
{
   if(!root)
   {
      return null;
   }
   else if(search_node(newkey, root, pos))
      return root;
   else
      return search(newkey, root->child[*pos], pos);
}

btnode *btree :: insert(int newdata, btnode *root)
{
   int meddata;
   btnode *medright, *newroot;
   if(push_down(newdata, root, &meddata, &medright))
   {
      newroot = new btnode;
      newroot->count = 1;
      newroot->data[1] = meddata;
      newroot->child[0] = root;
      newroot->child[1] = medright;
```

```
      return newroot;
   }
   return root;
}

int btree :: push_down(int newdata, btnode *curr, int
*meddata, btnode **medright)
{
   int pos;
   if(curr == null)
   {
      *meddata = newdata;
      *medright = null;
      return 1;
   }
   else
   {
      if(search_node(newdata, curr, &pos))
         cout << "\n\nError Duplicate Keys Cannot Be
         Inserted!!";
      if(push_down(newdata, curr->child[pos], meddata,
      medright))
         if(curr->count < max)
         {
            pushin(*meddata, *medright, curr, pos);
            return 0;
         }
         else
         {
            split_node(*meddata, *medright, curr, pos,
            meddata, medright);
            return 1;
         }
         return 0;
   }
}

void btree :: pushin(int meddata, btnode *medright,
btnode *curr, int pos)
{
   int p;
   for(p = curr->count; p > pos; p--)
```

```
    {
       curr->data[p + 1] = curr->data[p];
       curr->child[p + 1] = curr->child[p];
    }
    curr->data[pos + 1] = meddata;
    curr->child[pos + 1] = medright;
    curr->count++;
}

void btree :: split_node(int meddata, btnode *medright,
btnode *curr, int pos, int *newmedian, btnode *newright)
{
    int p, median;
    if(pos <= min)
    {
       median = min;
    }
    else
    {
       median = min + 1;
    }
    *newright = new btnode;
    for(p = median + 1; p <= max; p++)
    {
       (*newright)->data[p - median] = curr->data[p];
       (*newright)->child[p - median] = curr->child[p];
    }
    (*newright)->count = max - median;
    curr->count = median;
    if(pos <= min)
    {
       pushin(meddata, medright, curr, pos);
    }
    else
    {
       pushin(meddata, medright, *newright, pos - median);
    }
    *newmedian = curr->data[curr->count];
    (*newright)->child[0] = curr->child[curr->count];
    curr->count--;
}
```

```
void main()
{
   int ch, c, n;
   char ans;
   btree b;
   b.root = null;
   clrscr();
   do
   {
      cout << "\n\t\t>>>>>>>B-tree operations main
      menu<<<<<<<<<<<"
      <<"\n\n 1. Insert a key"
      <<"\n\n 2. Display the B-tree"
      <<"\n\n 3. Delete a key"
      <<"\n\n 4. Exit"
      <<"\n\n Enter choice:";
      ch = getche();
      ch = ch - '0';
      switch(ch)
      {
      case 1:
      do
      {
         cout << "\n\n\n\n Enter data:";
         cin >> n;
         b.root = b.insert(n, b.root);
        cout << "\n\n Do you want to insert more keys?";
         ans = getche();
         if(ans == 'n' || ans == 'N')
         break;
      }while(1);
      getch();
      break;
      case 2:
         b.pre_rec(b.root);
         getch();
      break;
      case 3:
         cout << "\n\n Enter key to be deleted: ";
         cin >> n;
         b.root = b.del(n, b.root);
         getch();
```

```
      break;
     case 4:
         exit(0);
         default:
         cout << "\n\tYou Have Entered An Invalid
         Choice!!!!!!!";
         getch();
      break;
   }
   }while(1);
}

/********************** OUTPUT **********************
 >>>>>>>B-tree operations main menu<<<<<<<<<<<
1. Insert a key
2. Display the B-tree
3. Delete a key
4. Exit
Enter choice: 1
Enter data:10
Do you want to insert more keys? y
Enter data: 20
Do you want to insert more keys?  y
Enter data: 30
Do you want to insert more keys? n
>>>>>>>B-tree operations main menu<<<<<<<<<<<
1. Insert a key
2. Display the B-tree
3. Delete a key
4. Exit
Enter choice: 2
   10   20   30
>>>>>>>B-tree operations main menu<<<<<<<<<<<
1. Insert a key
2. Display the B-tree
3. Delete a key
4. Exit
Enter choice: 1
Enter data: 40
Do you want to insert more keys? y
Enter data: 50
```

```
Do you want to insert more keys? n
>>>>>>>B-tree operations main menu<<<<<<<<<<<
1. Insert a key
2. Display the B-tree
3. Delete a key
4. Exit
Enter choice: 2
   30   10   20   40   50
>>>>>>>B-tree operations main menu<<<<<<<<<<<
1. Insert a key
2. Display the B-tree
3. Delete a key
4. Exit
Enter choice: 3
Enter key to be deleted: 10
>>>>>>>B-tree operations main menu<<<<<<<<<<<
1. Insert a key
2. Display the B-tree
3. Delete a key
4. Exit
Enter choice: 2
   20   30   40   50
```

13.3.3 B+ Tree

B+ trees are internal data structures. That is, the nodes contain whatever information is associated with the key as well as the key values. A variant of B-trees is often used as an *index tree*. A B+ tree combines the features of ISAM and B-trees as follows:

1. In an index tree, the pointers in the internal nodes point to other index nodes.
2. The pointers in the leaf nodes are not *nil*, but rather point to where the information associated with each key is stored on disk.
3. Each key must appear in a leaf node.
4. B-trees whose keys are only in the internal nodes of the tree and whose pointers in the leaf nodes print to where the related information is stored externally are called B+ trees.
5. Leaves are connected to form a linked list of keys in sequential order.
6. It has two parts—the index part consists of interior nodes and the sequence set consists of leaf nodes.
7. B+ trees are used to store index sequential file organization; the key values in the sequence set are the key values of record collections.

B+ Tree Structure

The structure of a B+ tree can be understood from the following points:

1. A B+ tree is in the form of a balanced tree where every path from the root of the tree to a leaf of the tree is of the same length.
2. Each non-leaf node (internal node) in the tree has between $\lceil n/2 \rceil$ and n children, where n is fixed.
3. The pointer (Ptr) can point to either a file record or a bucket of pointers so as to point to a file record.
4. Searching time is less in B+ trees but has some problem of wasted space.

Nodes of B+ Tree

A typical node structure of a B+ tree is shown in Fig. 13.14 with the nodes having the following characteristics:

1. Internal node of a B+ tree with $q - 1$ search values.
2. Leaf node of a B+ tree with $q - 1$ search values and $q - 1$ data pointers.

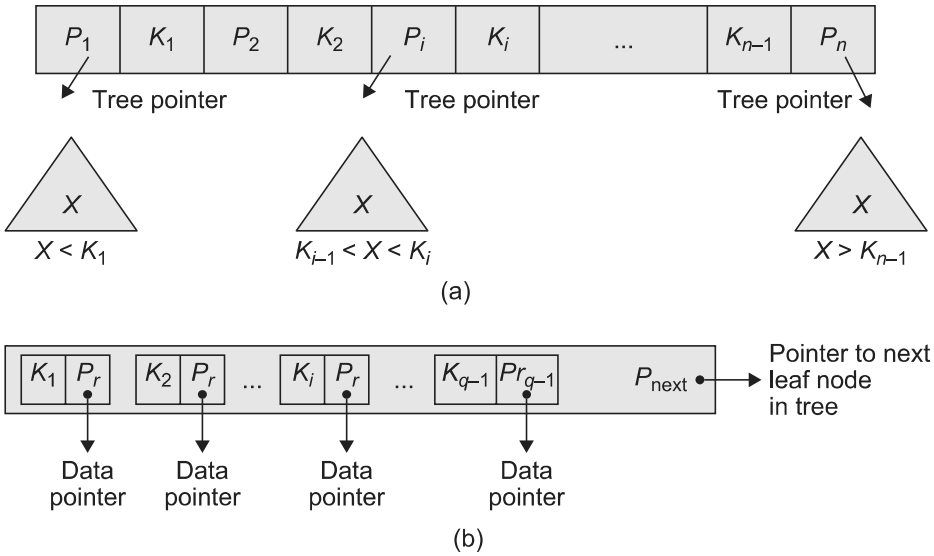


Fig. 13.14 Nodes of a B+ tree (a) Internal node of a B+ tree with $q - 1$ search values (b) Leaf node of a B+ tree with $q - 1$ search values and $q - 1$ data pointers

Non-leaf nodes form a multi-level sparse index on the leaf nodes.

1. Each leaf can hold up to $n - 1$ values and must contain at least $\lceil (n - 1)/2 \rceil$ values.
2. Non-leaf node pointers point to tree nodes (leaf nodes). Non-leaf nodes can hold up to n pointers and must hold at least $\lceil n/2 \rceil$ pointers.

This seemingly minor change has some major effects on the algorithms. For example, the leaf nodes and the internal nodes are treated differently when they split. When a leaf node splits, a copy of the middle key is moved up to be a separator at the next level. When an internal (index) node splits, the key itself is moved up to act as a separator.

Inserting nodes into a B+ tree The key value determines a record's placement in a B+ tree. The leaf nodes are maintained in sequential order and a doubly linked list (not shown) connects each leaf page with its sibling page(s). This doubly linked list speeds the data movement as the pages grow and contract.

We must consider three scenarios when we add a record to a B+ tree. Each scenario causes a different action.

1. If the leaf is not full and index (internal) is not full
   (a) Place the record in sorted position in the appropriate leaf node.
2. If the leaf is full and index is not full
   (a) Split the leaf node.
   (b) Place the middle key in the index node in sorted order.
   (c) Left leaf node contains records with keys below the middle key.
   (d) Right leaf node contains records with keys equal to or greater than the middle key.
3. If the leaf is full and index is full
   (a) Split the leaf node.
   (b) The records with keys < middle key go to the left leaf node.
   (c) The records with keys ≥ middle key go to the right leaf node.
   (d) Split the index node.
   (e) The keys < middle key go to the left index node.
   (f) The keys > middle key go to the right index node.
   (g) The middle key goes to the next (higher level) index.

If the next level index node is full, continue splitting the index node.

For example, inserting *a, d, g, f,* and *k* produces the B+ tree as in Fig. 13.15.

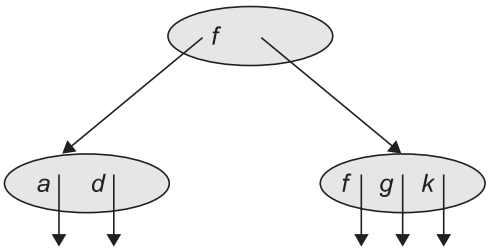


Fig. 13.15 Representation of insert operation

The arrow from the leaf nodes point to where the information associated with the key can be found.

Deleting nodes from a B+ tree The delete algorithm for B+ trees is listed as follows:

1. Leaf node not having keys < minimum keys and internal or index nodes not below the fill factor.

   Delete the record from the leaf page. Arrange keys in ascending order to fill `void`. If the key of the deleted record appears in the index page, use the next key to replace it.

2. Leaf node having keys < minimum keys and internal or index nodes not below the fill factor.

   Combine the leaf page and its sibling. Change the index page to reflect the change.

3. Leaf node having keys < minimum keys and internal or index nodes below the fill factor.
   (a) Combine the leaf page and its sibling.
   (b) Adjust the index page to reflect the change.
   (c) Combine the index page with its sibling.

Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

Another change is that the keys are deleted only from the leaf nodes. If a key to be deleted is also a part of the indexing structure (that is, appears in an internal node), it can remain in the index, for example, deleting *f* and *j* from the tree in Fig. 13.16 gives the B+ tree as in Fig. 13.17.

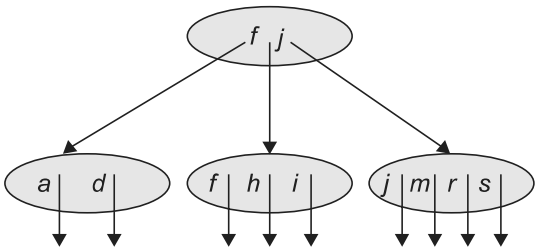


Fig. 13.16 Sample B+ tree

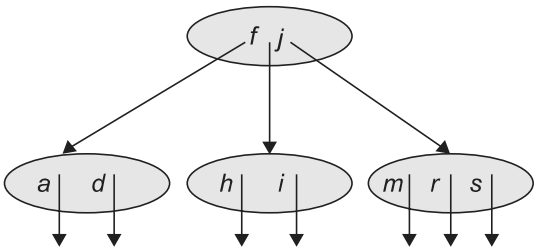


Fig. 13.17 Resultant tree after deletion of *f* and *i* from Fig. 13.16

The index says that all keys less than *f* are in the left subtree and those greater than or equal to *f* are in the right subtree. Likewise, all keys less than *j* are in the right subtree, all keys less than *j* are in the left subtree, and those greater than or equal to *g* are in the right subtree. This is still true.

Borrowing and coalescing are also slightly different because the old separator key can be discarded.

For example, deleting *g* from the B+ tree as in Fig. 13.18 leaves the leftmost node one key short.

Figure 13.19 is the result of borrowing *h*.

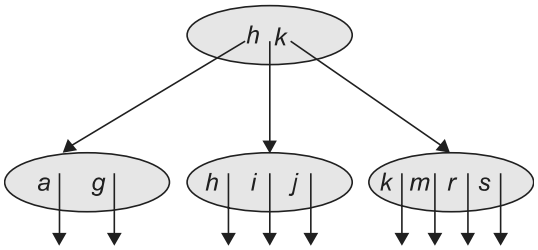


Fig. 13.18 Sample tree for deletion of *g*

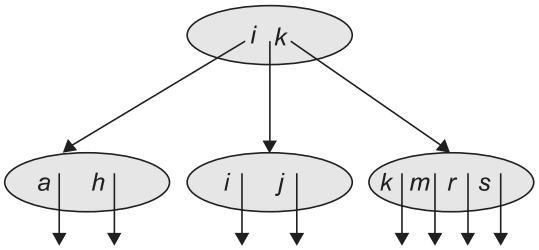


Fig. 13.19 Resultant tree after borrowing *h* from Fig. 13.18

Notice the difference in the B-tree. We rotated keys from one sibling to parent to other sibling. Here, the borrowed key goes directly into the sibling node and a copy of the new leftmost node becomes the separator in the parent. If we borrow from the left, however, a copy of the borrowed key becomes the separator.

If we coalesce, the keys *h*, *i*, and *j* go into the leftmost node. This is shown in Fig. 13.20.

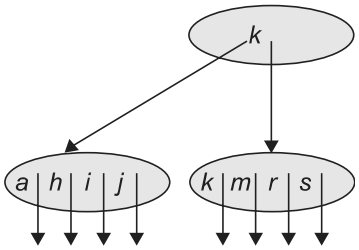


Fig. 13.20 Resultant tree after coalescing the keys *h*, *i*, and *j* into the leftmost node of Fig. 13.19

Since the pointers in the leaf nodes are not nil, we must have another way to recognize a leaf node. One way is to have a field in each node to mark the node as either an internal node or a leaf node. Another way is to continue using the leftmost child pointer as the flag because we need $M - 1$ pointers to point to the storage locations for $M - 1$ keys. By convention, we could let the pointer to the right of a key point to the data, and we could let the leftmost pointer be *nil* in a leaf node. This scheme handles the pointer consistently on insertion because the new pointer in the recursive call is stored to the right of the separator key being inserted.

If we do not use to the leftmost pointer to determine if we are at a leaf node, we can use it to link all the leaf nodes together. Having the leaf nodes linked together allows us to process the items in the file in order as well as access the items randomly via the index.

Advantages of B+ Trees over Indexed Sequential Access Method

The B+ tree is a dynamic index structure that adjusts gracefully to insertions and deletions It has the following advantages:

1. It is a balanced tree.
2. The leaf pages are not allocated sequentially. They are linked together through pointers (a doubly linked list).

13.3.4 Trie Tree

Instead of searching a tree using the entire key, we can consider the key to be a sequence of characters (letters or digits, for example), and use these characters to determine a multiway branch at each step. If we consider alphabetic keys, then we make a lexical

26-ary tree. At the first level, take a branch according to the first letter; at the second level of a tree, take a branch according to the second letter, and so on. If we consider the keys made up of three letters p, q, r, of maximum size 3, then the lexical tree will be 3-ary tree of level 3; nodes at first level having 3 pointers and 3 nodes at second level having 3 pointers each. So we get a total of 3×3 pointers at the second level and $3 \times 3 \times 3$ pointers at the third level. Finally, we store the actual key at a leaf.

The largest word determines the height of the lexical tree. So the drawback of the lexical tree is that after a few levels, it becomes very large. One solution is to prune from the tree all the branches that do not lead to any key. The resulting tree is called a *trie* (short for reTRIEvaL and pronounced 'try').

Consider a trie describing the words made only from the letters p, q, and r. The pruned branch can be shown as a null pointer marked with a cross in the node. Along with the branches to the next level of the trie, each node contains a data pointer to a key. Figure 13.21 shows a trie tree.

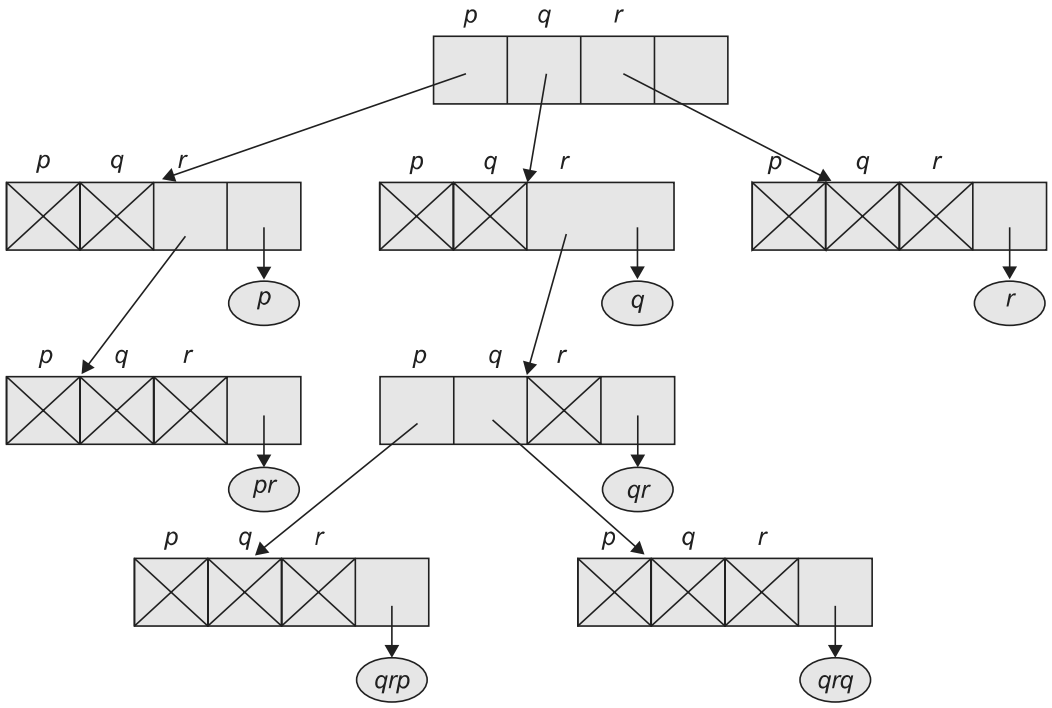


Fig. 13.21 Example of a trie tree

So the number of steps needed to search a trie is proportional to the number of characters in a key.

Declaration for Trie Tree

In each node of a trie tree, we have pointers to the next level and a pointer to the data. Program Code 13.16 is implementation of trie tree and the various operations that can be performed on it.

PROGRAM CODE 13.16

```
#define maxchar 3 /* Key is formed using 3 letters p,
q, r only */
#define max_key_length 5
class trieNode
{
   public:
      TrieNode *branch[maxchar];
      TrieData *dataptr;
};

typedef char key[max_key_length];
// SearchTrie() searches for the data starting from
// the root.
// If found, returns corresponding dataptr, otherwise
// returns null
TrieNode *SearchTrie(TrieNode *root, Key data)
{
   int p;
   for(p = 0; p < max_key_length&&root!=null; p++)
   {
      if(data[p]=='\0')
         break;
         /* data found, and root is pointing to the
         node having pointer to data */
      else
         root = root→branch[data[p] - 'p'];
   }
   if(root != null && root→dataptr == null)
      return null;
      return root;
}
```

13.3.5 Splay Tree

Of the many other variations on balanced binary trees, perhaps the most intriguing are the *splay trees*, introduced by Sleator and Tarjan, which are self-adjusting.

Splay trees are a form of BSTs. A splay tree maintains a balance without any explicit balance condition such as colour. Instead, 'splay operations', which involve rotations, are performed within the tree every time an access is made. The amortized cost of each operation on an n-node tree is $O(\log_2 n)$. One application of splay trees simplifies dynamic trees.

In an *amortized analysis*, the time required to perform a sequence of data structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive. Amortized analysis differs from the average case analysis where the probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.

Let us consider Example 13.1 to understand how indexing and search trees help in implementing practical applications efficiently.

EXAMPLE 13.1 Consider a hospital management system maintaining patient records. A patient who is currently in the hospital is said to be an active record, being consulted and updated continuously by attending physicians and nurses. When the patient leaves the hospital, the records become passive but still needed occasionally by the patient's physician or others. If, later, the patient is readmitted to the hospital, then the record becomes active again. The process of making such records active should be done faster.

Solution If we use a BST or even an AVL tree, then the records of the newly admitted patient's records will go to a leaf position, far from the root, and the access will be slower. Instead, we want to keep the records that are newly inserted or frequently accessed very near to the root, while the inactive records are kept far off, that is, in the leaf positions. However, we do not want to rebuild the tree into the desired shape. Instead, we need to make a tree a *self-adjusting data* structure that automatically changes its shape to bring the records closer to the root as they are used frequently, allowing inactive records to drift slowly down towards the leaves. Such trees are called as *splay trees*.

Splay trees are BSTs that achieve our goals by being self-adjusting in the following way: every time we access a node of the tree, whether for insertion or retrieval, we perform radical survey on the tree, lifting the newly accessed node all the way up so that it becomes the root of the modified tree. Other nodes are pushed out of the way as necessary to make space for this new root and not spacing them too far from the top position. Inactive nodes, on the other hand, will slowly be pushed farther and farther from the root.

13.3.6 Red–black Tree

A BST of height h can implement any of the basic dynamic set of operations in $O(h)$ time. Here, the operations are fast and the height of the search tree is small, but if its height is more, the performance may be no better than the linked list. *Red-black trees* are one of many search-tree schemes that are *balanced*. In order to guarantee that basic dynamic set operations, take $O(\log_2 n)$ time in the worst case.

Definition: A *red–black tree* is a BST with one extra bit of storage per node: its *colour*, which can either be red or black. Red–black trees were invented by R. Bayer under the same name 'symmetric binary B-trees'. Guibas and Sedgewick studied their properties at length and introduced the red/black colour convention.

The tree is balanced by constraining the way nodes can be coloured on any path from the root to a leaf; red-black tree ensures that no such path is more than twice as long as any other.

Properties of red–black trees Red–black trees have all the characteristics of BSTs. In addition, red-black trees have the following properties. In other words, a BST is a red-black tree if it satisfies the following properties.

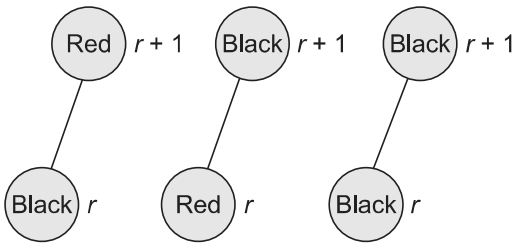
Each node of a tree contains these fields: colour, key, left, right, parent (and an optional field rank). If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value null.

1. Every node is either red or black.

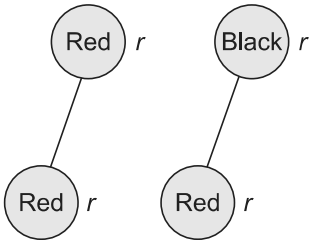
2. All the external nodes (leaf nodes) are black.
3. The rank in a tree goes from zero upto the maximum rank which occurs at the root. The rank of two consecutive nodes differs by utmost 1. Each leaf node has a rank 0.
4. If a node is red, then both its children are black. In other words, consecutive red nodes are disallowed. This means every red node is followed by a black node; on the other hand, a black node may be followed by a black or a red node. This implies that utmost 50% of the nodes on any path from external node to root are red.
5. The number of black nodes on any path from but not including the node x to leaf is called as *black height* of the node x, denoted as bh(x).

   Every simple path from the root to a leaf contains the same number of black nodes. In addition, every simple path from a node to a descendent leaf contains the same number of black nodes.

6. If a black node has a rank r, then its parent has the rank $r + 1$.



7. If a red node has a rank r, then its parent will have the rank r as well.



Example of a red–black tree Figure 13.22 is an example of a red–black tree with five levels. We have set ranks starting at the bottom.

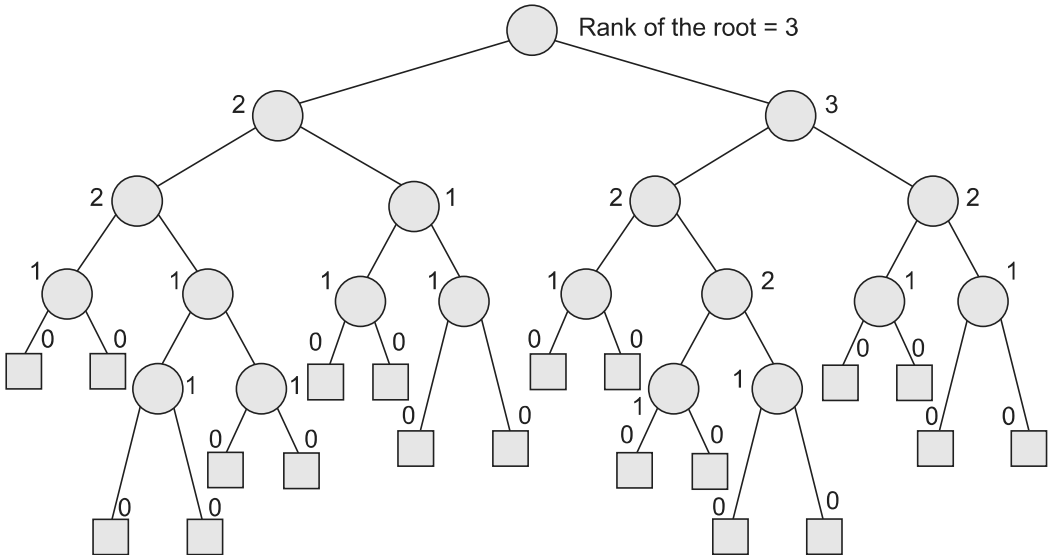


Fig. 13.22 Red–black tree

13.3.7 *K*-dimensional Tree

A *K*-dimensional tree (KD-tree*)* is a data structure used in computer science during orthogonal range searching, for instance, to find the set of points that fall into a given rectangle in a plane. Given a KD-tree of the points in question, it is possible to find the resulting points in O(sqrt(n) + k) time, where n is the number of points and k is the number of resultant points.

An example of KD-tree is shown in Fig. 13.23.

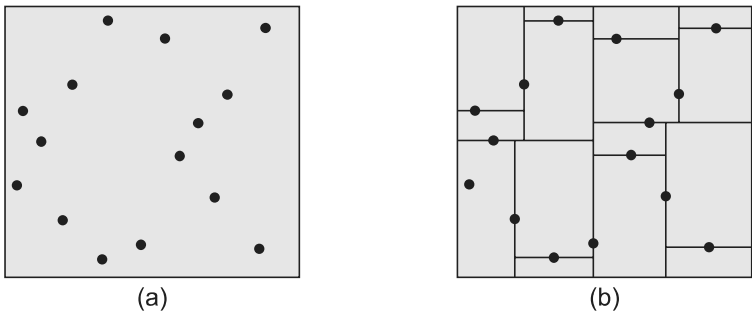


Fig. 13.23 KD-trees (a) Input (b) Output

Input description Let there be a set S of n points in k-dimensions.

Problem Construct a tree which partitions the space by half-planes such that each point is contained in its own region.

Although many different flavours of KD-trees have been devised, their purpose is always to hierarchically decompose space into a relatively small number of cells such that no cell contains too many input objects. This provides a fast way to access any input object by position. We traverse down the hierarchy until we find the cell containing the object and then scan through the few objects in the cell to identify the right one.

Typical algorithms construct KD-trees by partitioning point sets. Each node in the tree is defined by a plane through one of the dimensions that partition the set of points into left/right (or up/down) sets, each with half the points of the parent node. These children are again partitioned into equal halves, using places through a different dimension. Partitioning stops after log$n$ levels, with each point in its own leaf cell. Alternative KD-tree construction algorithms insert points incrementally and divide the appropriate cell although such trees can become seriously unbalanced.

A KD-tree can be constructed using Algorithm 13.1.

ALGORITHM 13.1

Input: A set of points P and *depth* the current depth
Output: The root of a KD-tree storing P

1. `if` P contains only one point `then`
2. `return` a leaf storing this point
3. `else if` depth is seven `then`
4. Split P into two subsets with a vertical line 1 through the median x-coordinate of the points in P. Let $P1$ be the set of points to the left and $P2$ be the set of points to the right. The points exactly on the line belong to $P1$
5. `else`
6. Split P into two subsets with a horizontal line 1 through the median y-coordinate of the points in P. Let $P1$ be the set of points above 1 and $P2$ be the points below 1. The points exactly on the line belong to $P1$
7. V_{right} = Build Kd-tree($P1$, depth + 1)
8. V_{left} = Build Kd-tree($P2$, depth + 1)
9. Create a node V with V_{right} and V_{left} as its right and left children, respectively
10. `return` V

This algorithm can be run in O($n$log$n$) time and uses O(n) storage.

The time constraint of O($n$log$n$) assumes that the median can be found in O(n) time. This is rather complicated in the general case but in our case can be made simply by presorting all the vertices in both x and y directions. Sorting takes O($n$log$n$) time and does therefore not worsen the time complexity of the overall algorithm.

13.3.8 AA Tree

We studied BSTs. A BST of n nodes is said to be balanced if the height is O(log$n$). A balanced tree supports efficient operations since most operations only have to traverse or

on two root-to-leaf paths. There are many implementations of balanced BSTs, including AVL trees, red–black trees, and AA trees. An AA tree is another alternative to AVL trees. An *AA tree* is a balanced BST with the following properties:

1. Every node is coloured either red or black.
2. The root is black.
3. If a node is red, both of its children are black.
4. Every path from a node to a null reference has the same number of black nodes.
5. Left children may not be red.

Figure 13.24 is an example of an AA tree.

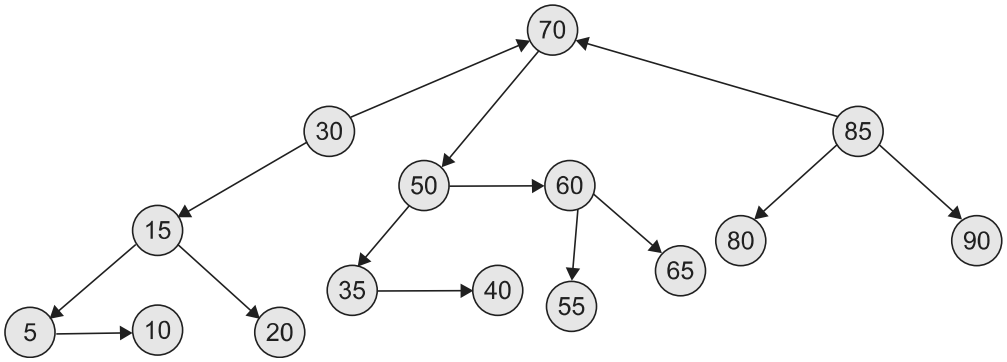


Fig. 13.24 AA tree

Advantages of AA Trees

AA trees are more advantageous as they simplify the algorithms. The following list explains the advantages:

1. They eliminate half the reconstructing cases.
2. They simplify deletion by removing an annoying case.
   (a) If an internal node has only one child, that child must be a red child.
   (b) We can always replace a node with the smallest child in the right subtree; it will either be a leaf node or have a red child.
3. An AA tree, which is a balanced BST, supports efficient operations, since most operations only have to traverse one or two root-to-leaf paths.

Representing Balance Information in AA Tree

In each node of AA tree, we store a *level*. The *level* is defined by the following rules:

1. If a node is a leaf, its level is one.
2. If a node is red, its level is the level of its parent.
3. If a node is black, its level is one less than the level of its parent.

Here, the *level* is the number of left links to a null reference.

Links in an AA tree A horizontal link is a connection between a node and a child with equal levels. The properties of such horizontal links are as follows:

1. Horizontal links are right references.
2. There cannot be two consecutives horizontal links.
3. Nodes at level two or higher must have two children.
4. If a node has no right horizontal link, its two children are at the same level.

RECAPITULATION

- A node of a BST has only one key value entry stored in it. A multiway tree has many key values stored in each node and thus each node may have multiple subtrees.
- Different indexing techniques are used to search a record in O(1) time. The index is a pair of key value and address. It is an indirect addressing that imposes order on a file without rearranging the file.
- Indexing techniques are classified as
  - Hashed indexing
  - Tree indexing
    - B-tree
    - B+ tree
    - Trie tree
- Splay trees are self-adjusting trees.

KEY TERMS

File organization A file is a collection of records, each record having one or more fields. The fields are used to distinguish among the records using keys. File organization is all about the way in which the records are stored in a file in an external storage media.

Index The index is a collection of pairs of the form (key value, address). It is an indirect addressing that imposes order on a file without rearranging it.

KD-tree A KD-tree is a data structure used in computer science during orthogonal range searching, for instance to find the set of points that fall into a given rectangle in a plane.

Multiway search tree In multiway search tree, there are 0 to m subtrees for each node, the node having k subtrees $(k \leq m)$ with k pointers and $k - 1$ value entries. The key values in the first subtree are all less than the key in the first entry; the key value in the other subtrees are all greater than or equal to the key in their parent entry.

Red–black tree A red-black tree is a BST with one extra bit of storage per node: its colour, which can either be red or black.

EXERCISES

Multiple choice questions

1. Which of the following remarks about the trie tree are false?
   Hint: More than one choice can be correct.
   (a) It is efficient in dealing with strings of variable length.
   (b) It is efficient if there are few number of data items.