Syllabus

MCA Sem-II, Paper - I

Data Structures

1. **Sorting and Searching Techniques**

   Bubble, Selection, Insertion, Shell sorts and Sequential, Binary, Indexed Sequential Searches, Interpolation, Binary Search Tree Sort, Heap sort, Radix sort.

   Analysis of Algorithms

   Algorithms, Pseudo code for expressing algorithms, time complexity and space complexity, O-notation, Omega notation and theta notation.

2. **Hashing Techniques**

   - Hash function
   - Address calculation techniques, Common hashing functions
   - Collision resolution
   - Linear probing, Quadratic
   - Double hashing
   - Bucket hashing
   - Deletion and rehashing

3. **Linear Lists**

   - Stacks: LIFO structure, create, POP, PUSH, delete stack
   - Queues: FIFO structure Priority Queues, Circular Queues, operations on Queues
   - Linear List Concept
   - List v/s Array, Internal pointer and External pointer, head, tail of a list, Null list, length of a list
   - Linked Lists
     - Nodes, Linked List Data Structure
   - Linked Lists algorithms
     - Create List
     - Insert Node (empty list, beginning, Middle, end)
     - Delete node (First, general case)
     - Search list
     - Retrieve Node, add node, Remove node, Print List

    - Append Linked List, array of Linked lists
- Complex Linked List structures
    - Header nodes
    - Circularly-Linked List
    - Doubly Linked List
        - Insertion, Deletion
    - Multilinked Lists
        - Insertion, Deletion

4. **Introduction to Trees**

- Binary Trees
    - Travesals (breadth-first, depth-first)
- Expression Trees
    - (Infix, Prefix, Postfix Traversals)
- General Trees
- Search Trees
- Binary Search Trees

5. **Heaps**

- Structure
- Basic algorithms – Reheap Up, Reheap Down, Build heap, Insert, Delete

6. **Multi-way Trees**

- M-way search trees
- B-Trees
    - Insertion (Inseet node, Search node, Split node, Insert entry)
    - Deletion (Node delete, Delete entry, Delete mid, ReFlow, Balance, Combine)
    - Traverse B-Tree
- B-Tree Search

- **Graphs**
    Terminology

- Operations (Add vertex, Delete vertex, Add Edge, Delete Edge, Find vertex)
- Traverse Graph (Depth-First, Breadth-First)
- Graph Storage Structures (Adjacency Matrix, Adjacency List)
- Networks
    - Minimum Spanning Tree
    - Shortest Path Algorithm
    - (Dijkstra's algorithm, Kruskal's algorithm, Prim's algorithm, Warshall's algorithm)

Term work/Practical: Each candidate will submit a journal / assignments in which at least 10 assignments based on the above syllabus along with the flow chart and program listing. Internal tests to be conducted separately.

1. Data structure – A Pseudocode Approach with C – Richard F Gilberg Behrouz A. Forouzan, Thomson.
2. Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2nd Edition.
3. Data structures and Program Design in C Robert Kruse, C.L. Tondo, Bruce Leung Pearson.
4. "Data structure using C" AM Tanenbaum, Y Langsam and MJ Augustein, Prentice Hall India.
5. "An Introduction to Structure with application" Jean – Paul Trembly and Paul Sorenson.
6. Data structure and Program design in CRL Kruse, BP Leung and CL Tondo Prentice-Hall.
7. Data structure and Algorithm Analysis in C Weiss, Mark Allen Addison Wesley.

Program List in Data Structures

1. Write a program in C to implement simple Stack, Queue, Circular Queue, Priority Queue.
2. Write a menu driven program that implements singly linked list for the following operations:

   Create, Display, Concate, merge, union, intersection.
3. Write a menu driven program that implements doubly linked list for the following operations:

   Create, Display, Count, Insert, Delete, Search, Copy, Reverse, Sort.
4. Write a menu driven program that implements doubly linked list for the following operations:

   Create, Display, Concate, merge, union, intersection.
5. Write a menu driven program that implements Singly circular linked list for the following operations:

   Create, Display, Count, Insert, Delete, Search, Copy, Reverse, Sort.
6. Write a program in C for sorting methods.
7. Write a menu driven program in C to

a. Create a binary search tree

b. Traverse the tree in Inorder, Preorder and Post Order

c. Search the tree for a given node and delete the node

Write a program in C to implement insertion and deletion in tree. B

8. Write a program in C to implement insertion and deletion in AVL tree.
9. Write a menu driven program that implements Heap tree (Maximum and Minimum Heap tree) for the following operations. (Using array) Insert, Delete.
10. Write a program to implement double hashing technique to map given key to the address space. Also write code for collision resolution (linear probing)
11. Write a program in C to implement Dijkstras shortest path algorithm for a given directed graph.
12. Write a program in C to insert and delete nodes in graph using adjacency matrix.
13. Write a program in C to implement Breadth First search using linked representation of graph.
14. Write a program in C to implement Depth first search using linked representation of graph.
15. Write a program in C to create a minimum spanning tree using Kruskal's algorithm.
16. Write a program in C to create a minimum spanning tree using Prim's algorithm.

1

SORTING AND SEARCHING TECHNIQUES

Unit Structure:

1.1 Sorting
1.2 Searching
1.3 Analysis of Algorithm
1.4 Complexity of Algorithm
1.5 Asymptotic Notations for Complexity Algorithms

1.1 SORTING

Sorting and searching are the most important techniques used in the computation. When the history of computing might be defined ‘searching‘ and ‘sorting’ would have been at top. They are the most common ingredients of programming systems. The sorting techniques are classified as internal and external sorting. Here we are going to discuss the different sorting techniques such as bubble sort, selection sort, Insertion, Shell sorts and Sequential, Binary, Indexed Sequential Searches, Interpolation, Binary Search ,Tree Sort, Heap sort, Radix sort.

1.1.1 Insertion Sort

It is one of the simplest sorting algorithms. It consists of N-1 passes. For pass P= 1 through N – 1, insertion sort ensures that the elements in positions 0 through P are sorted order. Insertion sort makes use of the fact that elements in positions 0 through P – 1 are already known to be in sorted order as shown in following table 1.1

Original	**34**	**8**	**64**	**51**	**32**	**21**	**Positions Moved**
P = 1	8	34	64	51	32	21	1
P = 2	8	34	64	51	32	21	0
P = 3	8	34	51	64	32	21	1
P = 4	8	32	34	51	64	21	3
P = 5	8	21	32	34	51	64	4

Table 1.1 Insertion sort after each pass

```
void insertionSort( int a[], int n ) {

/* Pre-condition: a contains n items to be sorted */

   int  i,j, p;

  /* Initially, the first item is considered 'sorted' */
   /* i divides a into a sorted region, x<i, and an
      unsorted one, x >= i */

   for(i=1;i<n;i++)
   {   /* Select the item at the beginning of the
          as yet unsorted section */

      p = a[i];
       /* Work backwards through the array, finding where v
          should go */
       j = i;
       /* If this element is greater than v,
            move it up one */
       while ( a[j-1] > p )
        {
         a[j] = a[j-1];
         j = j-1;
         if( j <= 0 ) break;
        }
       /* Stopped when a[j-1] <= p, so put p at position j */
       a[j] = p;
       }
   }
```

Figure 1.1 Function for insertion sort

Analysis of Insertion Sort

Due to the nested loops each of which can take N iterations, insertion sort is $O(N^2)$. In case if the input is pre-sorted, the running time is O(N), because the test in the inner loop always fails immediately. Indeed, if the input is almost sorted , insertion sort will run quickly.

1.1.2 Bubble Sort

This is common technique used to sort the elements. The bubble sort makes n – 1 passes through a sequence of n elements. Each pass moves through the array from left to right, comparing adjacent elements and swapping each pair that is out of order. This gradually moves the heaviest element to the right. It is called the bubble sort because if the elements are visualized in a vertical column, then each pass appears to 'bubble up' the next heaviest element by bouncing it off to smaller elements.

```
/* Bubble sort for integers */
#define SWAP(a,b)
 { int t; t=a; a=b; b=t; }

void bubble( int a[], int n )
/* Pre-condition: a contains n items to be sorted */
   {    int i, j;
   /* Make n passes through the array */
   for(i=0;i<n;i++)
     {        /* From the first element to the end of the unsorted
section */

      for(j=1;j<(n-i);j++)

        {  /* If adjacent items are out of order, swap them */
           if( a[j-1]>a[j] ) SWAP(a[j-1],a[j]);
         }
      }
   }
```

Figure 1.2 Bubble Sort

Analysis

Each of these algorithms requires ***n***-1 passes: each pass places one item in its correct place. (The n^{th} is then in the correct place also.) The i^{th} pass makes either ***i*** or ***n - i*** comparisons and moves.
The nth item also moves in its correct place. So:

$$T(n) = 1 + 2 + 3 + ... + (n-1)$$
$$= \sum_{i=1}^{n-1} i$$
$$= \frac{n}{2}(n-1)$$

or **O(n^2)** - but we already know we can use heaps to get an **O(*n* log*n*)** algorithm. Thus these algorithms are only suitable for small problems where their simple code makes them faster compared to the more complex code of the **O(*n* log*n*)** algorithm.

1.1.3 The Selection Sort

The selection sort is similar to the bubble sort. It makes the **n -1** passes through a sequence of n elements, each time moving the largest of the remaining unsorted elements into its correct position. This is more efficient than the Bubble sort because it doesn't move any element in the process of finding the largest element. It makes only one swap on each pass after it has found the largest. It is called the selection sort because on each pass it selects the largest of the remaining unsorted elements and puts it in its correct position.

```
void sort(int a[])
{  //Post condition: a[0] <= a[1] <= ... <= a[a.length -1] ;

   for(int i = n; i >0; i--)
{ int m=0;
   for(int j=0;j<=i;j++)
{   if(a[j] > a[m])
 {  m = j;}}

// Invariant: a[m] >= a[j] for all j<=i;
Swap(a,i,m)
// Invariants: a[j] <= a[i] for all j<= i;
                  a[i] <= a[i+1] <= .... <= n -1;
}}
```

Figure 1.3 Selection sort

Analysis

The Selection sort runs in O(n^2) time complexity. Though the Selection sort and Bubble sort have the same time complexity, selection sort is comparatively faster.

1.1.4 Shell Sort

This technique is invented by Donald shell. Compare to the Insertion sort, Shell sort uses the increment sequence, h1,h2,.....,ht. Any increment sequence will do as long as h1=1. After a phase,

using some increment hk, for every i, we have A[i] ≤ A[i + hk]. The file is then said to as hk-sorted.

Original	**81**	**94**	**11**	**96**	**12**	**35**	**17**	**95**	**28**	**58**	**41**	**75**	**15**
After5- sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Table 1.2 : Shell Sort flow

An important property of Shell Sort is that an hk-sorted file that is then hk-1 sorted remains hk-sorted. If this was not the case, the algorithm would be of little value, since work done by early phases would be undone by the later phases.

The general strategy to hk-sort is for each position, i, in hk, hk+1,.....,N – 1, place the element in the correct spot among i, i – hk , i - 2hk, etc. Although it doesnot affect the implementation, a careful examination shows that the action of an hk-sort is to perform an insertion sort on hk independent subarrays. A popular choice for increment sequence is to use the sequence suggested by shell:

h1= [N/2] and hk = [hk+1 / 2].

```
void Shellsort(int a[] , int N)
{
   int i,j, increment;
    int temp;
  for(increment = N/2 ; increment > 0; increment / = 2)
     for(i = increment; i < N ; i++)
   {
         temp = a[i];
          for(j = i ; jb>= increment ; j -= increment)
           if(temp < a[j - increment ])
             a[j] = a[j – increment];
          else
             break;
         a[j] = temp;
   }
}
```

Figure 1.4 Shell sort

Analysis
The Shell Sort runs in $O(n^{1.5})$ time.

1.1.5 Radix Sort

Bin Sort : The Bin sorting approach can be generalised in a technique that is known as *Radix sorting*.

An example

Assume that we have **n** integers in the range (**0,n^2**) to be sorted. (For a bin sort, **m** = **n**^{2}, and we would have an **O(n+m)** = **O(n^2)** algorithm.) Sort them in two phases:

1. Using **n** bins, place $\mathbf{a_i}$ into bin $\mathbf{a_i}$ mod **n**,
2. Repeat the process using **n** bins, placing $\mathbf{a_i}$ into bin floor($\mathbf{a_i}$/**n**), *being careful to append to the end of each bin*.

This results in a sorted list.

As an example, consider the list of integers:
36 9 0 25 1 49 64 16 81 4
n is 10 and the numbers all lie in (0,99). After the first phase, we will have:

Bin	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Content	0	181	-	-	644	25	3616	-	-	949

Table 1.3: Bins and contents

Note that in this phase, we placed each item in a bin indexed by the least significant decimal digit.

Repeating the process, will produce:

Bin	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Content	0149	16	25	36	49	-	64	-	81	-

Table 1.4: After reprocess

In this second phase, we used the leading decimal digit to allocate items to bins, being careful to add each item to the end of the bin.

We can apply this process to numbers of any size expressed to any suitable base or *radix*.

Generalised Radix Sorting

We can further observe that it's not necessary to use the same radix in each phase, suppose that the sorting key is a sequence of fields, each with bounded ranges, *eg* the key is a date using the structure:

```
typedef struct t_date {
   int day;
   int month;
   int year;
} date;
```

If the ranges for ***day*** and ***month*** are limited in the obvious way, and the range for ***year*** is suitably constrained, *eg* 1900 < **year** <= 2000, then we can apply the same procedure except that we'll employ a different number of bins in each phase. In all cases, we'll sort first using the least significant "digit" (where "digit" here means a field with a limited range), then using the next significant "digit", placing each item after all the items already in the bin, and so on.

Assume that the key of the item to be sorted has **k** fields, $\mathbf{f_i}|\mathbf{i}=0..\mathbf{k-1}$, and that each $\mathbf{f_i}$ has $\mathbf{s_i}$ discrete values, then a generalised radix sort procedure can be written:

radixsort(A, n) {
  for(i=0;i<k;i++) {
  for(j=0;j<s_i;j++) bin[j] = EMPTY;
	O(s_i)

for(j=0;j<n;j++) {
  move A_i
  to the end of bin[A_i->f_i]
  }
	O(n)

for(j=0;j<s_i;j++)
  concatenate bin[j] onto the end of A;
  }
}
	O(s_i)

Total	$\sum_{i=1}^{k} O(s_i + n) = O\left(kn + \sum_{i=1}^{k} s_i\right)$ $= O\left(n + \sum_{i=1}^{k} s_i\right)$

Figure 1.5 Radix sort

Now if, for example, the keys are integers in (0,$\mathbf{b^k}$-1), for some constant **k**, then the keys can be viewed as **k**-digit base-**b** integers. Thus, $\mathbf{s_i = b}$ for all **i** and the time complexity becomes **O(n+kb)** or **O(n)**. *This result depends on **k** being constant.*

If **k** is allowed to increase with **n**, then we have a different picture. For example, it takes $\log_2\mathbf{n}$ binary digits to represent an integer <**n**. If the key length were allowed to increase with **n**, so that **k** = log**n**, then we would have:

$$\sum_{i=1}^{k} O(s_i + n) = O\left(n \log n + \sum_{i=1}^{\log n} 2 \right)$$
$$= O(n \log n + 2\log n)$$
$$= O(n \log n)$$

Another way of looking at this is to note that if the range of the key is restricted to (0,$\mathbf{b^k}$-1), then we will be able to use the radixsort approach effectively if we allow duplicate keys when **n**>$\mathbf{b^k}$. However, if we need to have unique keys, then **k** must increase to at least $\log_b\mathbf{n}$. Thus, as **n** increases, we need to have log**n** phases, each taking **O(n)** time, and the radix sort is the same as quick sort!

1.1.6 The Heap Sort

A heap is by definition partially sorted, because each linear string from root to leaf is sorted. This leads to an efficient general sorting algorithm called the heap sort. Heap sort uses the auxillary function ***Sort()***. I has the complexity function ***O(n log n)*** which is same as merge and quick sort. The heap sort essentially loads n elements into a heap and then unloads them.

```
void sort()
{ // Post condition: a[0] <= a[1] <= ... < n-1;
   for(int i=n/2 - 1; i>= 0 ; i-- )
{  heapify(a,i,n);  }
for(int i = n – 1; i > 0; i--)
{ swap(a,0,i);
  heapify(a,0,i);
}
}
```

Figure 1.6 sort method for heap

```
void heapify(int a[], int i, int j)
{
    int a1 = a[i];
    while(2*i+1 < j)
 {   int k = 2*i + 1;
   if(k+1 < j && a[k+1] > a[k])
  {    ++k;  // a[k] is larger child
   }
 if(ai >= a[k])
{ break; }
a[i] = a[k];
i = k;
}  a[i] = ai;
 }
```

Figure 1.7 heapify method

The ***sort()*** function first converts the array so that its underlying complete binary tree is transformed into a heap. This is done by applying the heapify() function to each nontrivial subtree. The nontrivial subtrees are the subtrees that are rooted above the leaf level. In the array, the leaves are stored at **e** positions **a[n/2]** through **a[n].** So the first for loop in the sort() function applies the heapify() function to elements **a[n/2 - 1]** back through **a[0].** The result is an array whose corresponding tree has the heap property.
The main for loop progresses through **n-1** iterations, Each iteration does two things: it swaps the root element with element **a[i]** and then it applies the heapify() function to the subtree of elements **a[0..i].** That subtree consists of the part of the array that is still unsorted. Before swap executes on each iteration, the subarray **a[0..i]** has the heap property, so a[i] is the largest element in that subarray. That means that the swap() puts element a[i] in its correct position.

0	1	2	3	4	5	6	7	8
99	66	88	44	33	55	77	22	4

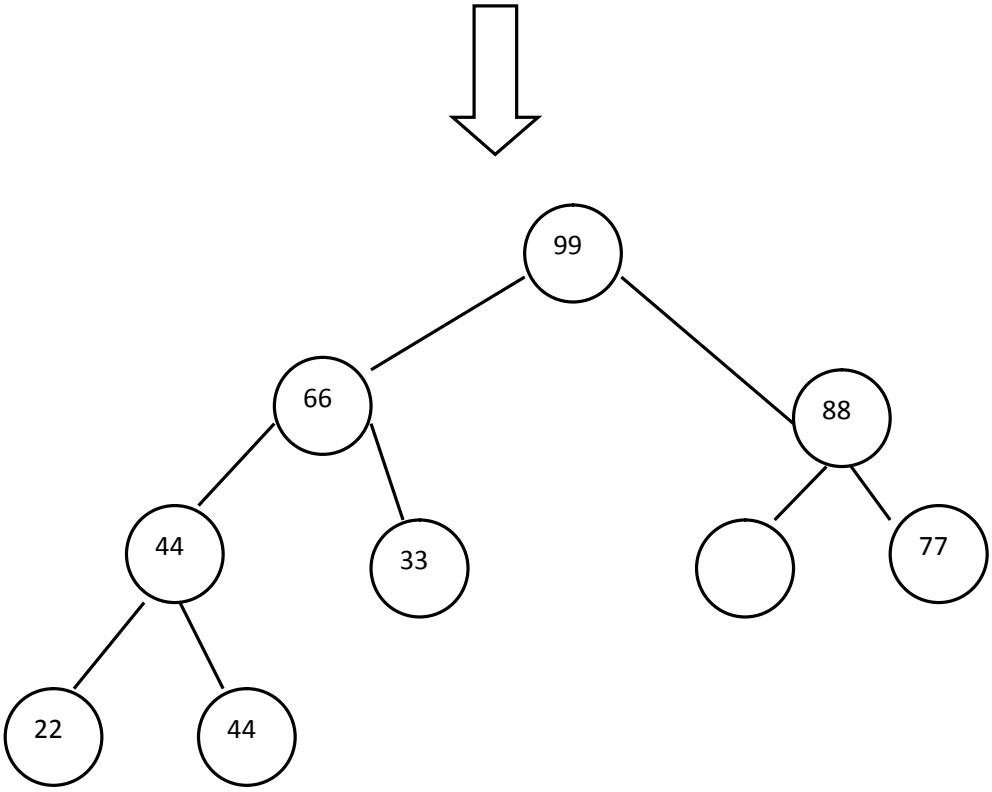


Figure 1.8 Implementation of heapify

1.2 SEARCHING

Searching technique is used to find a record from the set of records. The algorithm used for searching can return the entire record or it may return a pointer to that record. And it may also possible that the record may not exist in the given data set.

1.2.1Sequential Searching

It is the simplest form of search. This search is applicable to a table organized as an array or as a linked list. Let us assume that k is an array of n keys, k(0) through k(n-1), and r an array of records, r(0) through r(n-1), such that k(i) is the key of r(i). Also assume that key is a search argument. And we wish to return the smallest integer i such that k(i) equals key if such an i exists and -1 otherwise the algorithm for doing this is as follows

```
for(i = 0 ; i < n; i++)
    if(key == k(i))
      return(i);
return(-1);
```

As shown above the algorithm examines each key ; upon finding one that matches the search argument, its index is returned. If no match is found, -1 is returned.

The above algorithm can be modified easily to add record *rec* with the key *key* to the table if *key* is not already there.

```
k(n) = key;  /* insert the new key and record */
r(n) = rec;
n++;            /* increase the table size */
return(n - 1);
```

Compare to this more efficient search method involves inserting the argument key at the end of the array before beginning the search, thus guaranteeing that the key will be found.

```
k(n) = key;
for(i=0;key != k(i); i++)
if(i<n)
 return(i);
else
 return(-1);
```

for search and insertion, the entire if statement is replaced by

```
 if(i == n)
  r(n++) = rec;
return(i);
```

The extra key inserted at the end of the array is called a sentinel.

Efficiency of Sequential Searching

Through the example let us examine the number of comparisons made by a sequential search in searching for a given key. Let's assume no insertion or deletions exist, so that we are searching through a table of constant size n. The number of comparisons depends on where the record with the argument key appears in the table. If the record is the first one in the table, only one comparison is required; if the record is the last one in the table, n comparisons are required. If it is equally likely for the argument to

appear at any given table position, a successful search will take (on the average) (n+1) / 2 comparisons, and an unsuccessful search will take n comparisons. In any case, the number of comparisons is ***o(n)***.

1.2.2The Indexed Sequential Search

This is another technique which improves efficiency for a sorted file, but simultaneously it consumes a large amount of large space. As shown in the given figure an auxiliary table is called an ***index***, which is a set aside in addition to the sorted file itself. Each element in the index consists of a key ***kindex*** and a pointer to the record in the file that corresponds to *kindex*. The elements in the index, as well as the elements in the file, must be sorted on the key. If the index is **1/8th** the size of the file, every **8th** record of the file is represented in the index.

The execution of the algorithm is simple. Let r, k and key be defined as before, let *kindex* be an array of the keys in the index, and let *pindex* be the array of pointers within the index to the actual records in the file. Assume that the file is stored as an array **n** is the size of the file, and that indxsze is the size of the index.

```
for(i = 0; i < indxsize && kindex(i) <= key; i++)
lowlim = (i = 0) ? 0 : pindex(i-1);
hlim = (i == indxsize) ? n – 1:pindex(i) – 1;
for(j=lowlim ; j <= hilim && k(j) != key; j++)
return((j > hilim) ? -1 : j);
```

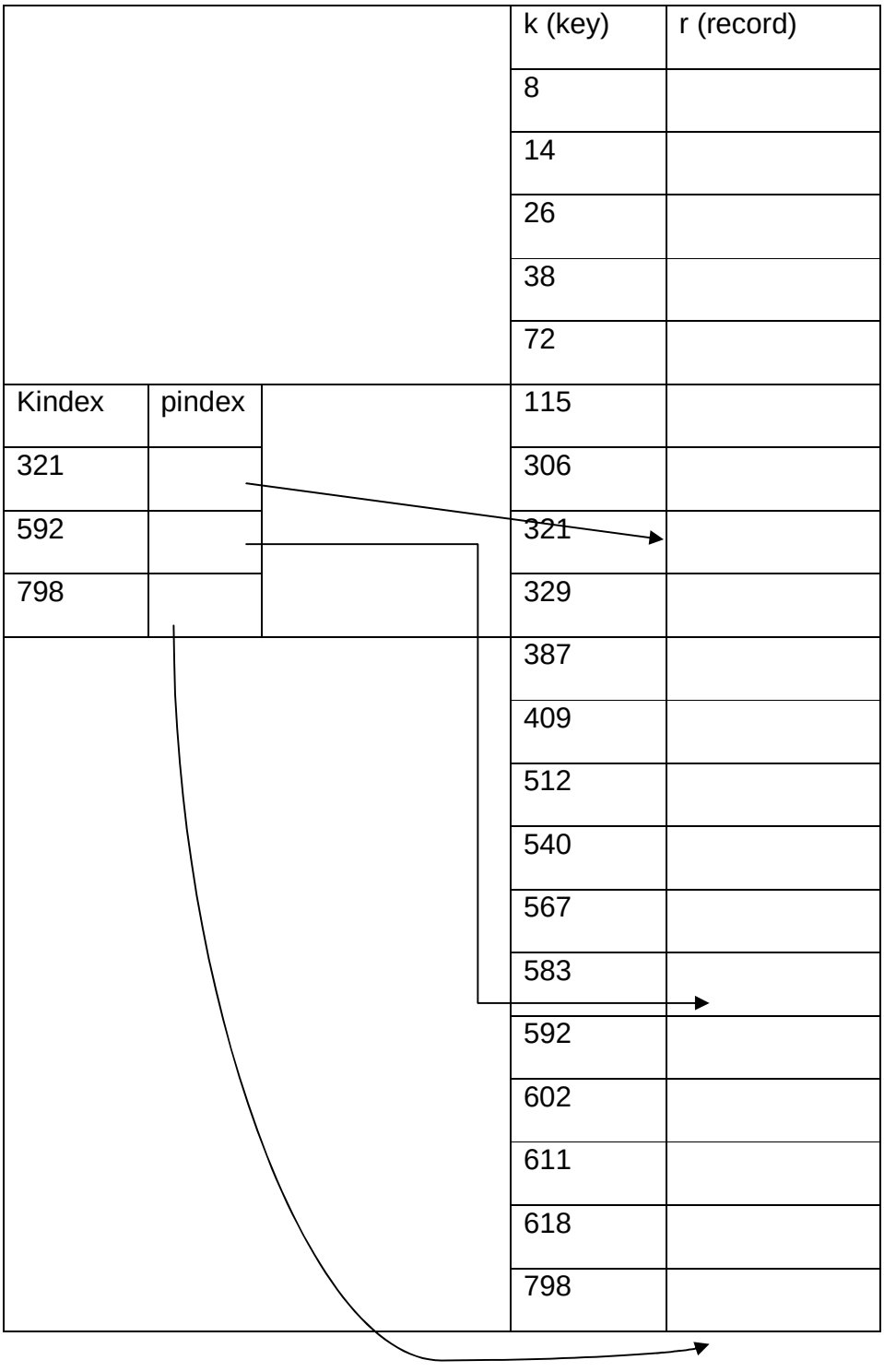


Figure 1.9 Indexing

The main advantage of indexed sequential method is that the items in the table can be examined sequentially if all the records in the file must be accessed, yet the search time for a particular item is sharply reduced. A sequential search is performed on the smaller index rather than on the larger table. And once the

correct index position has been found a second sequential search is performed on a small portion of the record table itself. Index is used for the stored array. If the table i so large that even the use of an index does not achieve sufficient efficiency, a secondary index can be used. The secondary index acts as an index to the primary index, which points to entries in the sequential table

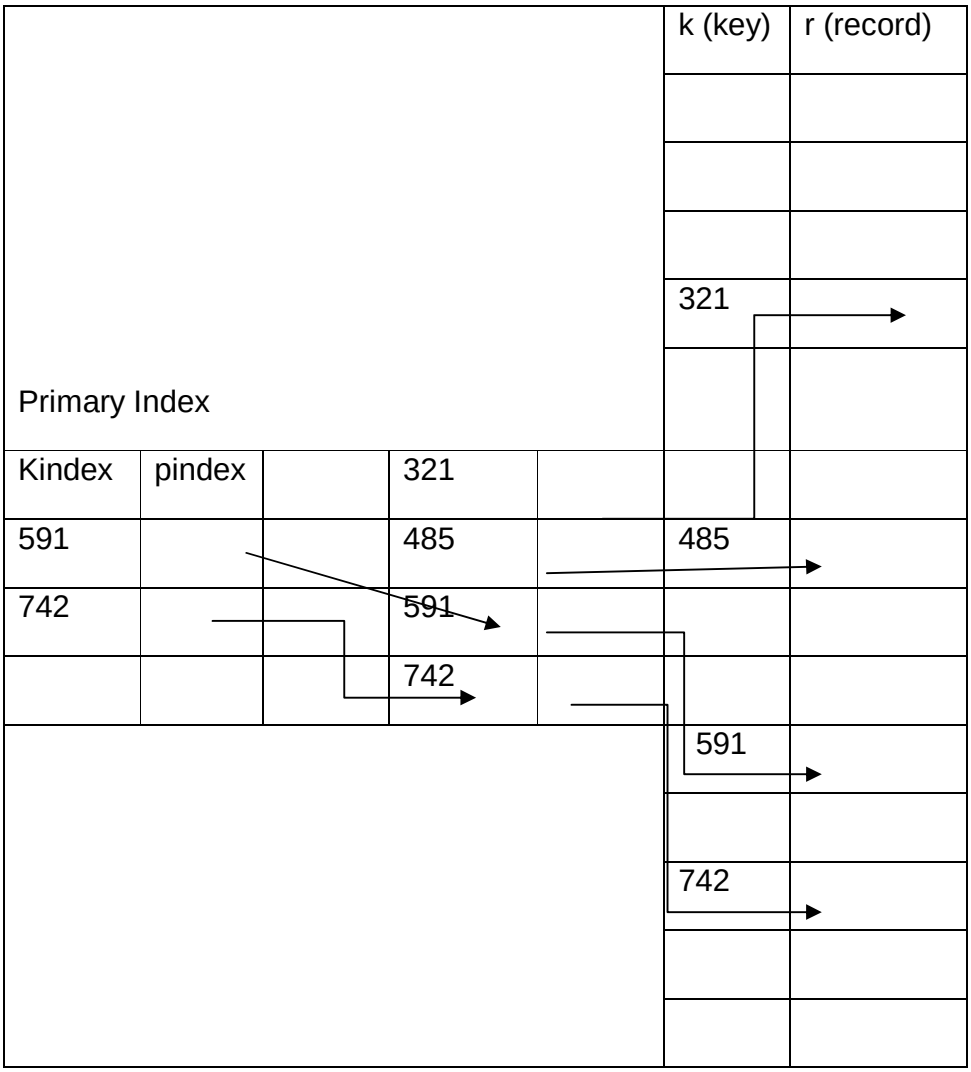


Figure 1.10 Indexed sequential

Deletions from an indexed sequential table can be made most easily by flagging deleted entries. In sequential searching through the table, deleted entries are ignored. Also note that if an element is deleted, even if its key is in the index, nothing need be done to the index, only the original table entry is flagged. Insertion into an sequential table is more difficult, since there may not be room between two already existing table entries.

1.2.3 Binary Search

This is the fastest approach of search. The concept is to look at the element in the middle. If the key is equal to that element, the search is finished. If the key is less than the middle element, do a binary search on the first half. If it's greater, do a binary search of the second half.

Working

The advantage of a binary search over a linear search is astounding for large numbers. For an array of a million elements, binary search, **O(log N),** will find the target element with a worst case of only 20 comparisons. Linear search, O(N), on average will take 500,000 comparisons to find the element. The only requirement for searching is that the array elements should be sorted.

```
int binarySearch(int sortedArray[], int first, int last, int key) {
  // function:
  //   Searches sortedArray[first]..sortedArray[last] for key.
  // returns: index of the matching element if it finds key,
  //         otherwise  -(index where it could be inserted)-1.
  // parameters:
  //   sortedArray in  array of sorted (ascending) values.
  //   first, last in  lower and upper subscript bounds
  //   key         in  value to search for.
  // returns:
  //    index of key, or -insertion_position -1 if key is not in the
array.       //  This value can easily be transformed into the
position to insert it.

  while (first <= last) {
     int mid = (first + last) / 2;  // compute mid point.
     if (key > sortedArray[mid])
        first = mid + 1;  // repeat search in top half.
     else if (key < sortedArray[mid])
        last = mid - 1; // repeat search in bottom half.
     else        return mid;     // found it. return position /////
  }
  return -(first + 1);    // failed to find key }
```

Figure 1.11 binary search flow

This is the most efficient method of searching a sequential table without the use of auxiliary indices or tables. This technique is further elaborated in next chapter.

1.3 ANALYSIS OF ALGORITHM

Algorithm : An algorithm is a systematic list of steps for solving a particular problem.

Analysis of algorithm is a major task in computer science. In order to compare algorithms, we must have some criteria to measure the efficiency of our algorithms. Suppose M is an algorithm and *n* is the size of the input data. The time and space used by the algorithm M are two main measures for the efficiency of M. The time is measured by counting the number of key operations in sorting and searching algorithms, i.e the number of comparisons.

1.4 COMPLEXITY OF ALGORITHMS

The *complexity* of an algorithm M is the function f(n) which gives the running time and / or storage space requirement of the algorithm in terms of the size n of the input data. Frequently ,the storage space required by an algorithm is simply a multiple of the data size n.

1.5 ASYMPTOTIC NOTATIONS FOR COMPLEXITY ALGORITHMS

The "big O" notation defines an upper bound function g(n) for f(n) which represents the time / space complexity of the algorithm on an input characteristic n. There are other asymptotic notations such as Ω, Θ, *o* which also serve to provide bounds for the function f(n).

1.5.1 Omega Notation (Ω)

The omega notation (Ω) is used when the function g(n) defines the lower bound for the function f(n).

Definition: f(n) = Ω(g(n)), iff there exists a positive integer n0 and a positive number M such that $|f(n)| \geq M|g(n)|$, for all $n \geq n0$.

For f(n) = 18n + 9, f(n) > 18n for all n, hence f(n) = Ω(n). Also, for $f(n) = 90n^2 + 18n + 6$, $f(n) > 90n^2$ for $n \geq 0$ and therefore $f(n) = \Omega(n^2)$.

For f(n) = Ω(g(n)), g(n) is a lower bound function and there may be several such functions, but it is appropriate that is the function which is almost as large a function of n is possible such that the definition of Ω is satisfied and is chosen as g(n).

1.5.2 Theta Notation (Θ)

The theta notation is used when the function f(n) is bounded both from above and below by the function g(n).

Definition: f(n) = **Θ** (g(n)), iff there exists a positive constants c1 and c2, and a positive integer n0 such that $c1|g(n)| \leq |f(n)| \leq c2|g(n)|$, for all $n \geq n0$.

From the definition it implies that the function g(n) is both an upper bound and a lower bound for the function f(n) for all values of n, $n \geq n0$. In other words,f(n) is such that, f(n)=O(g(n)) and f(n) = Ω(g(n))

For f(n) = 18n + 9, since f(n) > 18n and $f(n) \leq 27n$ for $n \geq 1$, we have f(n) = Ω(n) and therefore f(n) = O(n) respectively, for $n \geq 1$. Hence f(n) = **Θ** (n).

1.5.3 Small Oh Notation(*o*)

Definition: f(n)=o(g(n)) iff f(n)=O(g(n)) and f(n)≠ Ω(g(n)). For f(n)=18n+9, we have f(n)=O(n^2) but f(n)≠ Ω(n^2).Hence f(n) = o(n^2). However f(n) ≠ o(n).

Exercise:

1. Explain the insertion sort. Discuss its time and space complexity.

2. Sort the sequence 3,1,4,1,5,9,2,6,5 using insertion sort.

3. Explain Bubble sort with its analysis.

4. Explain the algorithm for shell sort.

5. Show the result of running Shellsort on the input 9,8,7,6,5,4,3,2,1 using increments {1,3,7}.

6. Write a program to implement selection algorithm.

7. Show how heapsort processes the input 142,543,123,65,453,879,572,434,111,242,811,102.

8. Using the radix sort, determine the running of radix sort.

9. Explain the sequential searching technique.

10. Explain the indexed sequential searching technique.

11. Give the comparison between sequential and indexed sequential.

12. Explain how binary searching is implemented. And also state its limitations.

13. What is time complexity?

14. Calculate the time complexity for binary searching.

15. Explain the following notations: a) big ‘Oh’ b) small ‘oh’ c) Theta (Θ) d) Omega Notation (Ω).

2

HASHING TECHNIQUES

Unit Structure:

2.1 Introduction
2.2 Hash Function
2.3 Open Hashing
2.4 Closed Hashing
2.5 Rehashing
2.6 Bucket Hashing

2.1 INTRODUCTION

The ideal hash table data structure is an array of some fixed size, containing the keys. Whereas, a key is a string. We will refer to the table size as *H_SIZE*, with the understanding that this is part of a hash data structure and not merely some variable floating around globally. The common convention is to have the table run from 0 to *H_SIZE-1.*

Each key is mapped into some number in the range 0 to *H_SIZE - 1* and placed in the appropriate cell. The mapping is called a *hash function*, which ideally should be simple to compute. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a Hash function that distributes the keys evenly among the cells. Figure 2.1 is an example, here *Raj* hashes to 3, Madhurya hashes to 4, *Rahul* hashes to 6, and *Priya* hashes to 7.

0	
1	
2	
3	Raj
4	Madhurya
5	
6	Rahul
7	Priya
8	
9	

Figure 2.1 An ideal hash table

This is the basic idea of hashing. The only remaining problems deal with choosing a function, deciding what to do when two keys hash to the same value (this is known as a *collision*), and deciding on the table size.

2.2. HASH FUNCTION

If the input keys are integers, then simply returning *key mod H_SIZE* (Table size) is generally a reasonable strategy, unless *key* happens to have some undesirable properties. In this case, the choice of hash function needs to be carefully considered. For instance, if the table size is 10 and the keys all end in zero, then the standard hash function is obviously a bad choice. When the input keys are random integers, then this function is not only very simple to compute but also distributes the keys evenly.

One option is to add up the ASCII values of the characters in the string. In Figure 2.2 we declare the type *INDEX*, which is returned by the hash function. The routine in Figure 2.3 implements this strategy and uses the typical C method of stepping through a string.

The hash function depicted in Figure 2.3 is simple to implement and computes an answer quickly. However, if the table size is large, the function does not distribute the keys well. For instance, suppose that *H_SIZE* = 10,007 (10,007 is a prime number). Suppose all the keys are eight or fewer characters long. Since a *char* has an integer value that is always at most 127, the hash function can only assume values between 0 and 1016, which is 127 * 8. This is clearly not an equitable distribution!

```
typedef unsigned int INDEX;
```

Figure 2.2 Type returned by hash function

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;
/*1*/       while( *key != '\0' )
/*2*/           hash_val += *key++;
/*3*/       return( hash_val % H_SIZE );
}
```

Figure 2.3 A simple hash function

Another hash function is shown in Figure 2.4. This hash function assumes *key* has at least two characters plus the NULL terminator. 27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2. This function only examines the first three characters, but if these are random, and the table size is 10,007, as before, then we would expect a reasonably equitable distribution. Although there are $26^3 = 17{,}576$ possible combinations of three characters (ignoring blanks), a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only 2,851. Even if none of *these* combinations collide, only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonably large.

Figure 2.5 shows a third attempt at a hash function. This hash function involves all characters in the key and can generally be expected to distribute well (it computes $\sum_{i=0}^{KeySize - 1} key[key\_size - i - 1] \, 32^{i,}$ and brings the result into proper range). The code computes a polynomial function (of 32) by use of Horner's rule. For instance, another way of computing $h_k = k_1 + 27k_2 + 27^2k_3$ is by the formula $h_{k =} ((k_3) * 27 + k_2) * 27 + k_1$. Horner's rule extends this to an *n*th degree polynomial.

We have used 32 instead of 27, because multiplication by 32 is not really a multiplication, but amounts to bit-shifting by five. In line 2, the addition could be replaced with a bitwise exclusive or, for increased speed.

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
return ( ( key[0] + 27*key[1] + 729*key[2] ) % H_SIZE );
}
```

Figure 2.4 Another possible hash function -- not too good

```
INDEX
hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = O;
/*1*/       while( *key != '\0' )
/*2*/           hash_val = ( hash_val << 5 ) + *key++;
/*3*/       return( hash_val % H_SIZE );
}
```

Figure 2.5 A good hash function

The hash function described in Figure 2.5 is not necessarily the best with respect to table distribution, but does have the merit of extreme simplicity (and speed if overflows are allowed). If the keys are very long, the hash function will take too long to compute. Furthermore, the early characters will wind up being left-shifted out of the eventual answer. A common practice in this case is not to use all the characters. The length and properties of the keys would then influence the choice. For instance, the keys could be a complete street address. The hash function might include a couple of characters from the street address and perhaps a couple of characters from the city name and ZIP code. Some programmers implement their hash function by using only the characters in the odd spaces, with the idea that the time saved computing the hash function will make up for a slightly less evenly distributed function.

The main programming detail left is collision resolution. If, when inserting an element, it hashes to the same value as an already inserted element, then we have a *collision* and need to resolve it. There are several methods for dealing with this. We will discuss two of the simplest: open hashing and closed hashing.*

*These are also commonly known as separate chaining and open addressing, respectively.

2.3. OPEN HASHING (SEPARATE CHAINING)

The first strategy, commonly known as either *open hashing,* or *separate chaining*, is to keep a list of all elements that hash to the same value. We assume that the keys are the first 10 perfect squares and that the hashing function is simply *hash*(*x*) = *x mod* 10. (The table size is not prime, but is used here for simplicity.) Figure 2.6 should make this clear.

To perform a *find*, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the item is found. To perform an *insert*, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Sometimes new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

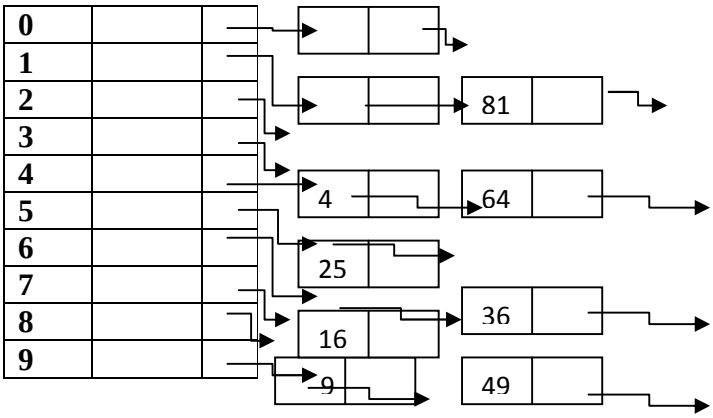


Figure 2.6 An open hash table

The type declarations required to implement open hashing are in Figure 2.7. The hash table structure contains the actual size and an array of linked lists, which are dynamically allocated when the table is initialized. The *HASH_TABLE* type is just a pointer to this structure.

```
typedef struct list_node *node_ptr;
struct list_node
{
element_type element;
node_ptr next;
};
typedef node_ptr LIST;
typedef node_ptr position;
/* LIST *the_list will be an array of lists, allocated later */
/* The lists will use headers, allocated later */
struct hash_tbl
{
unsigned int table_size;
LIST *the_lists;
};
typedef struct hash_tbl *HASH_TABLE;
```

Figure 2.7 Type declaration for open hash table

Notice that the the_lists field is actually a pointer to a pointer to a list_node structure. If typedefs and abstraction are not used, this can be quite confusing.

```
HASH_TABLE
initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
/*1*/       if( table size < MIN_TABLE_SIZE )
{
/*2*/           error("Table size too small");
/*3*/           return NULL;
}
/* Allocate table */
/*4*/       H = (HASH_TABLE) malloc ( sizeof (struct hash_tbl) );
/*5*/       if( H == NULL )
/*6*/           fatal_error("Out of space!!!");
/*7*/       H->table_size = next_prime( table_size );
/* Allocate list pointers */
/*8*/           H->the_lists = (position *) malloc ( sizeof (LIST) * H-
>table_size );
/*9*/       if( H->the_lists == NULL )
/*10*/           fatal_error("Out of space!!!");
/* Allocate list headers */
/*11*/      for(i=0; i<H->table_size; i++ )
{
/*12*/           H->the_lists[i] = (LIST) malloc ( sizeof (struct list_node)
);
/*13*/           if( H->the_lists[i] == NULL )
/*14*/               fatal_error("Out of space!!!");
else
/*15*/               H->the_lists[i]->next = NULL;
}
/*16*/      return H;
}
```

Figure 2.8 Initialization routine for open hash table

Figure 2.8 shows the initialization function, which uses the same ideas that were seen in the array implementation of stacks. Lines 4

through 6 allocate a hash table structure. If space is available, then H will point to a structure containing an integer and a pointer to a list. Line 7 sets the table size to a prime number, and lines 8 through 10 attempts to allocate an array of lists. Since a LIST is defined to be a pointer, the result is an array of pointers.

If our LIST implementation was not using headers, we could stop here. Since our implementation uses headers, we must allocate one header per list and set its next field to NULL. This is done in lines 11 through 15. Also lines 12 through 15 could be replaced with the statement

H->the_lists[i] = make_null();

Since we have not used this option, because in this instance it is preferable to make the code as self-contained as possible, it is certainly worth considering. An inefficiency of our code is that the malloc on line 12 is performed H->table_size times. This can be avoided by replacing line 12 with one call to malloc before the loop occurs:

H->the lists = (LIST*) malloc

(H->table_size * sizeof (struct list_node));

Line 16 returns H.

The call find(key, H) will return a pointer to the cell containing key. The code to implement this is shown in Figure 2.9.

Next comes the insertion routine. If the item to be inserted is already present, then we do nothing; otherwise we place it at the front of the list (see Fig. 2.10).*

```
position

find( element_type key, HASH_TABLE H )

{

position p;

LIST L;

/*1*/       L = H->the_lists[ hash( key, H->table_size) ];

/*2*/       p = L->next;

/*3*/       while( (p != NULL) && (p->element != key) )
```

```
/* Probably need strcmp!! */
/*4*/           p = p->next;
/*5*/       return p;
}
```

Figure 2.9 Find routine for open hash table

```
void insert( element_type key, HASH_TABLE H )
{
position pos, new_cell;
LIST L;
/*1*/     pos = find( key, H );
/*2*/     if( pos == NULL )
{
/*3*/         new_cell = (position) malloc(sizeof(struct list_node));
/*4*/         if( new_cell == NULL )
/*5*/             fatal_error("Out of space!!!");
else
{
/*6*/             L = H->the_lists[ hash( key, H->table size ) ];
/*7*/             new_cell->next = L->next;
/*8*/             new_cell->element = key; /* Probably need strcpy!! */
/*9*/             L->next = new_cell;
}
}
}
```

Figure 2.10 Insert routine for open hash table

The element can be placed anywhere in the list; this is most convenient in our case. The insertion routine coded in Figure 2.10 is somewhat poorly coded, because it computes the hash function twice. Redundant calculations are always bad, so this code should be rewritten if it turns out that the hash routines account for a significant portion of a program's running time.

The deletion routine is a straightforward implementation of deletion in a linked list, so we will not bother with it here. Any scheme could be used besides linked lists to resolve the collisions- a binary search tree or even another hash table would work.

We define the load factor of a hash table to be the ratio of the number of elements in the hash table to the table size. The effort required to perform a search is the constant time required to evaluate the hash function plus the time to traverse the list.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Closed hash table with linear probing, after each insertion

In an unsuccessful search, the number of links to traverse is (excluding the final NULL link) on average. A successful search requires that about 1 + (1/2) links be traversed, since there is a guarantee that one link must be traversed (since the search is successful), and we also expect to go halfway down a list to find our match. This analysis shows that the table size is not really important, but the load factor is. The general rule for open hashing is to make the table size about as large as the number of elements expected.

2.4. CLOSED HASHING (OPEN ADDRESSING)

Open hashing has the limitation of requiring pointers. This will slow down the algorithm a bit because of the time required to allocate new cells, and also essentially requires the implementation of a second data structure. Closed hashing, also known as open addressing, is an alternative to resolving collisions with linked lists. In a closed hashing system, if a collision occurs, alternate cells are

tried until an empty cell is found. More formally, cells h0(x), h1(x), h2(x), . . . are tried in succession, where

hi(x) = (hash(x) + F(i))mod H_SIZE, with F(0) = 0.

The function, F, is the collision resolution strategy. Because all the data goes inside the table, a bigger table is needed for closed hashing than for open hashing. Generally, the load factor should be below $\lambda = 0.5$ for closed hashing.

The following are the three common collision resolution strategies.

Linear Probing

Quadratic Probing

Double Hashing

2.4.1 Linear Probing

In linear probing, F is a linear function of i, typically F(i) = i. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Figure 2.11 shows the result of inserting keys {89, 18, 49, 58, 69} into a hash table using the same hash function as before and the collision resolution strategy, F(i) = i.

The first collision occurs when 49 is inserted; it is put in spot 0 which is open. key 58 collides with 18, 89, and then 49 before an empty cell is found three away. The collision for 69 is handled in a similar manner. As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Worse, even if the table is relatively empty, blocks of occupied cells start forming. This effect is known as **Primary Clustering.**

We will assume a very large table and that each probe is independent of the previous probes. These assumptions are satisfied by a random collision resolution strategy and are reasonable unless λ is very close to 1. First, we derive the expected number of probes in an unsuccessful search. This is just the expected number of probes until we find an empty cell. Since the fraction of empty cells is $1 - \lambda$, the number of cells we expect to probe is $1/(1 - \lambda)$. The number of probes for a successful search is equal to the number of probes required when the particular element was inserted. When an element is inserted, it is done as a result of an unsuccessful search. Thus we can use the cost of an

unsuccessful search to compute the average cost of a successful search.

The caveat is that ƛ changes from 0 to its current value, so that earlier insertions are cheaper and should bring the average down. For instance, in the table above, ƛ= 0.5, but the cost of accessing 18 is determined when 18 is inserted. At that point, ƛ = 0.2. Since 18, was inserted into a relatively empty table, accessing it should be easier than accessing a recently inserted element such as 69. We can estimate the average by using an integral to calculate the mean value of the insertion time, obtaining

$$I(ƛ) = 1/ ƛ \int_0^{ƛ} 1/ 1 - x \, dx = 1 / ƛ \ln 1 / 1 - ƛ$$

These formulas are clearly better than the corresponding formulas for linear probing. Clustering is not only a theoretical problem but actually occurs in real implementations. Figure 2.12 compares the performance of linear probing (dashed curves) with what would be expected from more random collision resolution. Successful searches are indicated by an S, and unsuccessful searches and insertions are marked with U and I, respectively.

$$h(x) + i^2 = h(x) + j^2 \quad (mod\ H\_SIZE)$$

$$i^2 = j^2 \quad (mod\ H\_SIZE)$$

$$i^2 - j^2 = 0 \quad (mod\ H\_SIZE)$$

$$(i - j)(i + j) = 0 \quad (mod\ H\_SIZE)$$

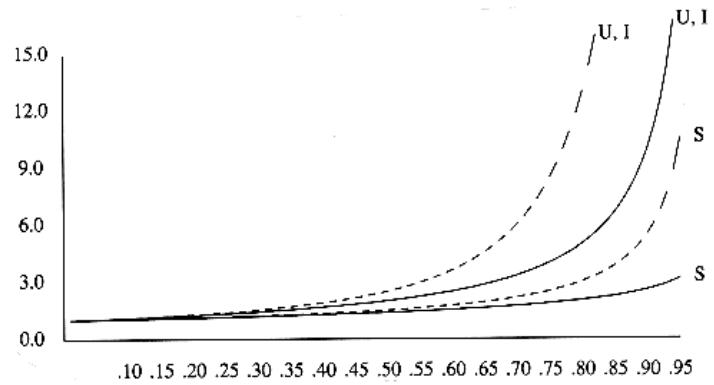


Figure 2.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy. S is successful search,U is unsuccessful search, I is insertion

If ƛ = 0.75, then the formula above indicates that 8.5 probes are expected for an insertion in linear probing. If ƛ = 0.9, then 50 probes are expected, which is unreasonable. This compares with 4 and 10 probes for the respective load factors if clustering were not a problem. We see from these formulas that linear probing can be a bad idea if the table is expected to be more than half full. If ƛ = 0.5, however, only 2.5 probes are required on average for insertion and only 1.5 probes are required, on average, for a successful search.

2.4.2. Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadratic probing is what you would expect-the collision function is quadratic. The popular choice is F(i) = i2. Figure 2.13 shows the resulting closed table with this collision function on the same input used in the linear probing example.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 2.13 Closed hash table with quadratic probing, after each insertion

When 49 collides with 89, the next position attempted is one cell away. This cell is empty, so 49 is placed there. Next 58 collides at position 8. Then the cell one away is tried but another collision occurs. A vacant cell is found at the next cell tried, which is 22 = 4 away. 58 is thus placed in cell 2. The same thing happens for 69.

For quadratic probing, the situation is even more drastic: There is no guarantee of finding an empty cell once the table gets

more than half full, or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternate locations to resolve collisions.

Indeed, we prove now that if the table is half empty and the table size is prime, then we are always guaranteed to be able to insert a new element.

THEOREM 2.1.

If quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

PROOF:

Let the table size, H-SIZE, be an (odd) prime greater than 3. We show that the first [H-SIZE/2] alternate locations are all distinct. Two of these locations are h(x) + i2(mod H-SIZE) and h(x) + j2(mod H-SIZE), where 0 < i, j <[H-SIZE/2]. Suppose, for the sake of contradiction, that these locations are the same, but i ≠ j. Then

h(x) + i2 = h(x) + j2 (mod H_SIZE)
i2 = j2 (mod H_SIZE)
i2 - j2 = 0 (mod H_SIZE)(i - j)(i + j) = 0 (mod H_SIZE)

Since H_SIZE is prime, it follows that either (i - j) or (i + j) is equal to 0 (mod H_SIZE). Since i and j are distinct, the first option is not possible.

Since 0 < i, j <[H_SIZE/2], the second option is also impossible. Thus, the first [H_SIZE/2] alternate locations are distinct. Since the element to be inserted can also be placed in the cell to which it hashes (if there are no collisions), any element has [H_SIZE/2] locations into which it can go. If at most [H_SIZE/2] positions are taken, then an empty spot can always be found.

If the table is even one more than half full, the insertion could fail (although this is extremely unlikely). Therefore, it is important to keep this in mind. It is also crucial that the table size be prime.* If the table size is not prime, the number of alternate locations can be severely reduced. As an example, if the table size were 16, then the only alternate locations would be at distances 1, 4, or 9 away.

*If the table size is a prime of the form 4k + 3, and the quadratic collision resolution strategy f(i) = + i2 is used, then the entire table can be probed. The cost is a slightly more complicated routine.

Standard deletion cannot be performed in a closed hash table, because the cell might have caused a collision to go past it. For instance, if we remove 89, then virtually all of the remaining finds will fail. Thus, closed hash tables require lazy deletion, although in this case there really is no laziness implied.

```
enum kind_of_entry { legitimate, empty, deleted };
struct hash_entry
{
element_type element;
enum kind_of_entry info;
};
typedef INDEX position;
typedef struct hash_entry cell;
/* the_cells is an array of hash_entry cells, allocated later */
struct hash_tbl
{
unsigned int table_size;
cell *the_cells;
};
typedef struct hash_tbl *HASH_TABLE;
```

Figure 2.14 Type declaration for closed hash tables

```
HASH_TABLE
initialize_table( unsigned int table_size )
{
HASH_TABLE H;
int i;
/*1*/       if( table_size < MIN_TABLE_SIZE )
{
/*2*/           error("Table size too small");
/*3*/           return NULL;
```

```
}
/* Allocate table */
/*4*/       H = (HASH_TABLE) malloc( sizeof ( struct hash_tbl ) );
/*5*/       if( H == NULL )
/*6*/           fatal_error("Out of space!!!");
/*7*/       H->table_size = next_prime( table_size );
/* Allocate cells */
/*8*/       H->the cells = (cell *) malloc
( sizeof ( cell ) * H->table_size );
/*9*/       if( H->the_cells == NULL )
/*10*/          fatal_error("Out of space!!!");
/*11*/      for(i=0; i<H->table_size; i++ )
/*12*/          H->the_cells[i].info = empty;
/*13*/      return H;
}
```

Figure 2.15 Routine to initialize closed hash table

As with open hashing, find(key, H) will return the position of key in the hash table. If key is not present, then find will return the last cell. This cell is where key would be inserted if needed. Further, because it is marked empty, it is easy to tell that the find failed. We assume for convenience that the hash table is at least twice as large as the number of elements in the table, so quadratic resolution will always work. Otherwise, we would need to test i before line 4. In the implementation in Figure 2.16, elements that are marked as deleted count as being in the table. Lines 4 through 6 represent the fast way of doing quadratic resolution. From the definition of the quadratic resolution function, f(i) = f(i - 1) + 2i -1, so the next cell to try can be determined with a multiplication by two (really a bit shift) and a decrement. If the new location is past the array, it can be put back in range by subtracting H_SIZE. This is faster than the obvious method, because it avoids the multiplication and division that seem to be required. The variable name i is not the best one to use; we only use it to be consistent with the text.

```
position

find( element_type key, HASH_TABLE H )
{
position i, current_pos;
```

```
/*1*/      i = 0;
/*2*/      current_pos = hash( key, H->table_size );
/* Probably need strcmp! */
/*3*/      while( (H->the_cells[current_pos].element != key ) &&
(H->the_cells[current_pos].info != empty ) )
{
/*4*/          current_pos += 2*(++i) - 1;
/*5*/          if( current_pos >= H->table_size )
/*6*/              current_pos -= H->table_size;
}
/*7*/      return current_pos;
}
```

Figure 2.16 Find routine for closed hashing with quadratic probing

The final routine is insertion. As with open hashing, we do nothing if key is already present. It is a simple modification to do something else. Otherwise, we place it at the spot suggested by the find routine. The code is shown in Figure 2.17.

2.4.3. Double Hashing

The last collision resolution method we will examine is double hashing. For double hashing, one popular choice is F(i) = i.h2(x). This formula says that we apply a second hash function to x and probe at a distance h2(x), 2h2(x), . . ., and so on. For instance, the obvious choice h2(x) = x mod 9 would not help if 99 were inserted into the input in the previous examples. Thus, the function must never evaluate to zero. It is also important to make sure all cells can be probed (this is not possible in the example below, because the table size is not prime). A function such as h2(x) = R - (x mod R), with R a prime smaller than H_SIZE, will work well. If we choose R = 7, then Figure 2.18 shows the results of inserting the same keys as before.

```
void insert( element_type key, HASH_TABLE H )
{
position pos;
pos = find( key, H );
if( H->the_cells[pos].info != legitimate )
```

```
{    /* ok to insert here */
H->the_cells[pos].info = legitimate;
H->the_cells[pos].element = key;
/* Probably need strcpy!! */
}
}
```

Figure 2.17 Insert routine for closed hash tables with quadratic probing

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 2.18 Closed hash table with double hashing, after each insertion

The first collision occurs when 49 is inserted. h2(49) = 7 - 0 = 7, so 49 is inserted in position 6. h2(58) = 7 - 2 = 5, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance h2(69) = 7 - 6 = 1 away. If we tried to insert 60 in position 0, we would have a collision. Since h2(60) = 7 - 4 = 3, we would then try positions 3, 6, 9, and then 2 until an empty spot is found. It is generally possible to find some bad case, but there are not too many here.

As we have said before, the size of our sample hash table is not prime. We have done this for convenience in computing the hash function, but it is worth seeing why it is important to make sure the table size is prime when double hashing is used. If we attempt to insert 23 into the table, it would collide with 58. Since h2(23) = 7 -

2 = 5, and the table size is 10, we essentially have only one alternate location, and it is already taken. Thus, if the table size is not prime, it is possible to run out of alternate locations prematurely. However, if double hashing is correctly implemented, simulations imply that the expected number of probes is almost the same as for a random collision resolution strategy. This makes double hashing theoretically interesting. Quadratic probing, however, does not require the use of a second hash function and is thus likely to be simpler and faster in practice.

2.5. REHASHING

If the table gets too full, the running time for the operations will start taking too long and inserts might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions. A solution for this is to build another table that is about twice as big and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.

For example, suppose the elements 13, 15, 24, and 6 are inserted into a closed hash table of size 7. The hash function is h(x) = x mod 7. Suppose linear probing is used to resolve collisions. The resulting hash table appears in Figure 2.19.

If 23 is inserted into the table, the resulting table in Figure 2.20 will be over 70 percent full. Because the table is so full, a new table is created. The size of this table is 17, because this is the first prime which is twice as large as the old table size. The new hash function is then h(x) = x mod 17. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table. The resulting table appears in Figure 2.21. This entire operation is called rehashing. This is obviously a very expensive operation -- the running time is O(n), since there are n elements to rehash and the table size is roughly 2n, but it is actually not all that bad, because it happens very infrequently. In particular, there must have been n/2 inserts prior to the last rehash, so it essentially adds a constant cost to each insertion.

0	6
1	15
2	
3	24
4	
5	
6	13

Figure 2.19 Closed hash table with linear probing with input 13,15, 6, 24

0	6
1	15
2	23
3	24
4	
5	
6	13

Figure 2.20 Closed hash table with linear probing after 23 is inserted

0	
1	
2	
3	
4	
5	

6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Figure 2.21 Closed hash table after rehashing

Rehashing can be implemented in several ways with Quadratic probing. One alternative is to rehash as soon as the table is half full. The other extreme is to rehash only when an insertion fails. A third, middle of the road, strategy is to rehash when the table reaches a certain load factor. Rehashing frees the programmer from worrying about the table size and is important because hash tables cannot be made arbitrarily large in complex programs.

```
HASH_TABLE

rehash( HASH_TABLE H )

{

unsigned int i, old_size;

cell *old_cells;

/*1*/       old_cells = H->the_cells;

/*2*/       old_size = H->table_size;

/* Get a new, empty table */
```

```
/*3*/       H = initialize_table( 2*old_size );
/* Scan through old table, reinserting into new */
/*4*/       for( i=0; i<old_size; i++ )
/*5*/           if( old_cells[i].info == legitimate )
/*6*/               insert( old_cells[i].element, H );
/*7*/       free( old_cells );
/*8*/       return H;
}
```

Figure 2.22 Shows that rehashing is simple to implement.

2.6 BUCKET HASHING

Closed hashing stores all records directly in the hash table. Each record *R* with key value k_R has a **home position** that is **h**(k_R), the slot computed by the hash function. If *R* is to be inserted and another record already occupies *R*'s home position, then *R* will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be. Naturally, the same policy must be followed during search as during insertion, so that any record not found in its home position can be recovered by repeating the collision resolution process.

One implementation for closed hashing groups hash table slots into **buckets**. The *M* slots of the hash table are divided into *B* buckets, with each bucket consisting of *M/B* slots. The hash function assigns each record to the first slot within one of the buckets. If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an **overflow bucket** of infinite capacity at the end of the table. All buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket.

When searching for a record, the first step is to hash the key to determine which bucket should contain the record. The records in this bucket are then searched. If the desired key value is not found and the bucket still has free slots, then the search is complete. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in

the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

Exercises

1. Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = x(mod 10), show the resulting

a. open hash table

b. closed hash table using linear probing

c. closed hash table using quadratic probing

d. closed hash table with second hash function h2(x) = 7 - (x mod 7)

2. Show the result of rehashing the hash tables in above input data.

3. Write a program to compute the number of collisions required in a long random sequence of insertions using linear probing, quadratic probing, and double hashing.

4. A large number of deletions in an open hash table can cause the table to be fairly empty, which wastes space. In this case, we can rehash to a table half as large. Assume that we rehash to a larger table when there are twice as many elements as the table size. How empty should an open table be before we rehash to a smaller table?

5. An alternative collision resolution strategy is to define a sequence, f(i) = ri, where r0 = 0 and r1, r2, . . . , rn is a random permutation of the first n integers (each integer appears exactly once).

a. Prove that under this strategy, if the table is not full, then the collision can always be resolved.

b. Would this strategy be expected to eliminate clustering?

c. If the load factor of the table is λ, what is the expected time to perform an insert?

d. If the load factor of the table is λ, what is the expected time for a successful search?

e. Give an efficient algorithm (theoretically as well as practically) to generate the random sequence. Explain why the rules for choosing P are important.

6. What are the advantages and disadvantages of the various collision resolution strategies?

7. A spelling checker reads an input file and prints out all words not in some online dictionary. Suppose the dictionary contains 30,000 words and the file is one megabyte, so that the algorithm can make only one pass through the input file. A simple strategy is to read the dictionary into a hash table and look for each input word as it is read. Assuming that an average word is seven characters and that it is possible to store words of length l in l + 1 bytes (so space waste is not much of a consideration), and assuming a closed table, how much space does this require?

8. If memory is limited and the entire dictionary cannot be stored in a hash table, we can still get an efficient algorithm that almost always works. We declare an array H_TABLE of bits (initialized to zeros) from 0 to TABLE_SIZE - 1. As we read in a word, we set H_TABLE[hash(word)] = 1. Which of the following is true?

a. If a word hashes to a location with value 0, the word is not in the dictionary.

b. If a word hashes to a location with value 1, then the word is in the dictionary.

Suppose we choose TABLE_SIZE = 300,007.

c. How much memory does this require?

d. What is the probability of an error in this algorithm?

e. A typical document might have about three actual misspellings per page of 500 words. Is this algorithm usable?

9. *Describe a procedure that avoids initializing a hash table (at the expense of memory).

10. Suppose we want to find the first occurrence of a string p1p2. . . pk in a long input string a1a2 . . . an. We can solve this problem by hashing the pattern string, obtaining a hash value hp, and comparing this value with the hash value formed from a1a2 . . . ak,a2a3 . . . ak+1,a3a4 . . . ak+2, and so on until an-k+1an-k+2 . . . an. If we have a match of hash values, we compare the strings

character by character to verify the match. We return the position (in a) if the strings actually do match, and we continue in the unlikely event that the match is false.

a. Show that if the hash value of aiai+1 . . . ai + k - 1 is known, then the hash value of ai+1ai+2 . . . ai+k can be computed in constant time.

b. Show that the running time is O(k + n) plus the time spent refuting false matches.

c. Show that the expected number of false matches is negligible.

d. Write a program to implement this algorithm.

e. Describe an algorithm that runs in O(k + n) worst case time.

f. Describe an algorithm that runs in O(n/k) average time.

11. Write a note on bucket hashing.

3

STACKS, QUEUES AND LINKED LISTS

Unit Structure:

3.1 Abstract Data Types (ADTS)
3.2 The List Abstract Data Type
3.3 The Stack ADT
3.4 The Queue ADT

3.1 ABSTRACT DATA TYPES (ADTS):

One of the basic rules concerning programming is that no routine should ever exceed a page. This is accomplished by breaking the program down into *modules*. Each module is a logical unit and does a specific job. Its size is kept small by calling other modules. Modularity has several advantages.

(a) it is much easier to debug small routines than large routines.

(b) it is easier for several people to work on a modular program simultaneously.

(c) a well-written modular program places certain dependencies in only one routine, making changes easier.

For instance, if output needs to be written in a certain format, it is certainly important to have one routine to do this. If printing statements are scattered throughout the program, it will take considerably longer to make modifications.

The idea that global variables and side effects are bad is directly attributable to the idea that modularity is good.

An *abstract data type* (ADT) is a set of operations. Abstract data types are mathematical abstractions; nowhere in an ADT's definition is there any mention of *how* the set of operations is implemented. This can be viewed as an extension of modular design.

Objects such as lists, sets, and graphs, along with their operations, can be viewed as abstract data types, just as integers, reals, and booleans are data types. Integers, reals, and booleans have operations associated with them, and so do abstract data

types. For the set ADT, we might have such operations as *union, intersection, size*, and *complement*. Alternately, we might only want the two operations *union* and *find,* which would define a different ADT on the set.

The basic idea is that the implementation of these operations is written once in the program, and any other part of the program that needs to perform an operation on the ADT can do so by calling the appropriate function. If for some reason implementation details need to change, it should be easy to do so by merely changing the routines that perform the ADT operations. This change, in a perfect world, would be completely transparent to the rest of the program.

There is no rule telling us which operations must be supported for each ADT; this is a design decision. Error handling and tie breaking (where appropriate) are also generally up to the program designer. The three data structures that we will study in this chapter are primary examples of ADTs. We will see how each can be implemented in several ways, but if they are done correctly, the programs that use them will not need to know which implementation was used.

3.2 THE LIST ABSTRACT DATA TYPE:

We will deal with a general list of the form a_1, a_2, a_3, . . . , a_n. We say that the size of this list is n. We will call the special list of size 0 a *null list*.

For any list except the null list, we say that a_{i+1} follows (or succeeds) a_i ($i < n$) and that a_{i-1} precedes a_i ($i > 1$). The first element of the list is a_1, and the last element is a_n. We will not define the predecessor of a_1 or the successor of a_n. The *position* of element a_i in a list is i. Throughout this discussion, we will assume, to simplify matters, that the elements in the list are integers, but in general, arbitrarily complex elements are allowed.

Associated with these "definitions" is a set of operations that we would like to perform on the list ADT. Some popular operations are

(a) *print_list* and *make_null*, which do the obvious things;

(b) *find*, which returns the position of the first occurrence of a key;

(c) *insert* and *delete*, which generally insert and delete some key from some position in the list; and

(d) *find_kth*, which returns the element in some position (specified as an argument).

If the list is 34, 12, 52, 16, 12, then *find*(52) might return 3; *insert*(*x*,3) might make the list into 34, 12, 52, *x*, 16, 12 (if we insert

after the position given); and *delete*(3) might turn that list into 34, 12, *x*, 16, 12.

Of course, the interpretation of what is appropriate for a function is entirely up to the programmer, as is the handling of special cases (for example, what does *find*(1) return above?). We could also add operations such as *next* and *previous*, which would take a position as argument and return the position of the successor and predecessor, respectively.

3.2.1. Simple Array Implementation of Lists

Obviously all of these instructions can be implemented just by using an array. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, especially if there are many lists of unknown size.

An array implementation allows *print_list* and *find* to be carried out in linear time, which is as good as can be expected, and the *find_kth* operation takes constant time. However, insertion and deletion are expensive. For example, inserting at position 0 (which amounts to making a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is $O(n)$. On average, half the list needs to be moved for either operation, so linear time is still required. Merely building a list by n successive inserts would require quadratic time.

Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

3.2.2. Linked Lists

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved. Figure 3.1 shows the general idea of a *linked list*.

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the *next* pointer. The last cell's *next* pointer points to *; this value is defined by C and cannot be confused with another pointer. ANSI C specifies that* is zero.

Recall that a pointer variable is just a variable that contains the address where some other data is stored. Thus, if *p* is declared to be a pointer to a structure, then the value stored in *p* is interpreted as the location, in main memory, where a structure can be found. A field of that structure can be accessed by *p->field_name*, where *field_name* is the name of the field we wish to examine. Figure 3.2 shows the actual representation of the list in Figure 3.1. The list contains five structures, which happen to reside in memory locations 1000, 800, 712, 992, and 692 respectively. The *next* pointer in the first structure has the value 800, which provides the indication of where the second structure is. The other structures each have a pointer that serves a similar purpose. Of course, in order to access this list, we need to know where the first cell can be found. A pointer variable can be used for this purpose. It is important to remember that a pointer is just a number. For the rest of this chapter, we will draw pointers with arrows, because they are more illustrative.

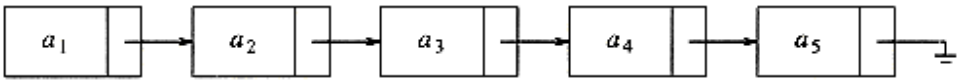


Figure 3.1 A linked list

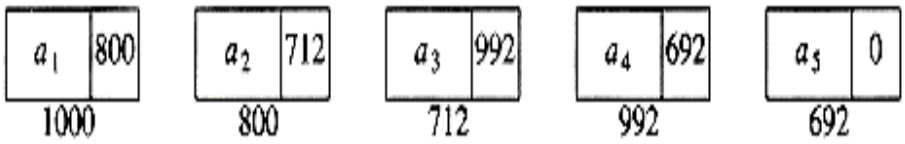


Figure 3.2 Linked list with actual pointer values

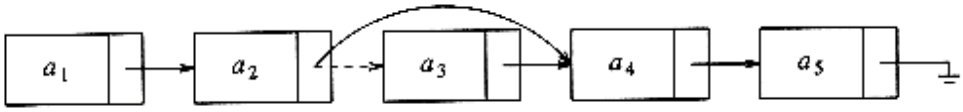


Figure 3.3 Deletion from a linked list

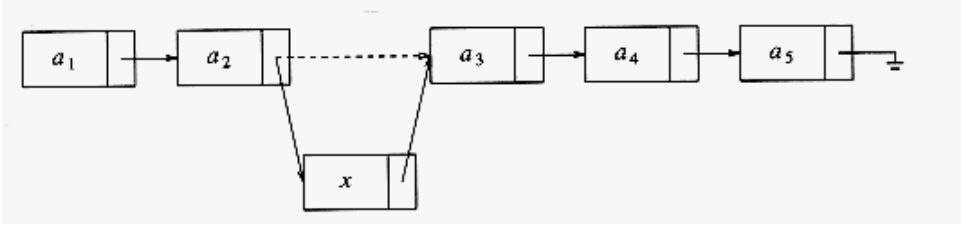


Figure 3.4 Insertion into a linked list

To execute *print_list(L)* or *find(L,key)*, we merely pass a pointer to the first element in the list and then traverse the list by following the *next* pointers. This operation is clearly linear-time, although the constant is likely to be larger than if an array implementation were used. The *find_kth* operation is no longer quite as efficient as an array implementation; *find_kth(L,i)* takes $O(i)$ time and works by traversing down the list in the obvious manner. In practice, this bound is pessimistic, because frequently the calls to *find_kth* are in sorted order (by i). As an example, *find_kth(L,2), find_kth(L,3), find_kth(L,4), find_kth(L,6)* can all be executed in one scan down the list.

The *delete* command can be executed in one pointer change. Figure 3.3 shows the result of deleting the third element in the original list.

The *insert* command requires obtaining a new cell from the system by using an *malloc* call (more on this later) and then executing two pointer maneuvers. The general idea is shown in Figure 3.4. The dashed line represents the old pointer.

3.2.3. Programming Linked Lists

The description above is actually enough to get everything working, but there are several places where you are likely to go wrong. First of all, there is no really obvious way to insert at the front of the list from the definitions given. Second, deleting from the front of the list is a special case, because it changes the start of the list; careless coding will lose the list. A third problem concerns deletion in general. Although the pointer moves above are simple, the deletion algorithm requires us to keep track of the cell *before* the one that we want to delete.

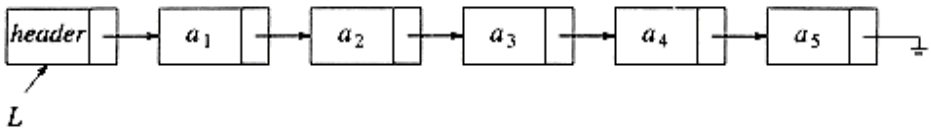


Figure 3.5 Linked list with a header

It turns out that one simple change solves all three problems. We will keep a sentinel node, which is sometimes referred to as a *header* or *dummy* node. This is a common practice, which we will see several times in the future. Our convention will be that the header is in position 0. Figure 3.5 shows a linked list with a header representing the list $a_1, a_2, \ldots, a_5$.

To avoid the problems associated with deletions, we need to write a routine *find_previous*, which will return the position of the predecessor of the cell we wish to delete. If we use a header, then

if we wish to delete the first element in the list, *find_previous* will return the position of the header. The use of a header node is somewhat controversial. Some people argue that avoiding special cases is not sufficient justification for adding fictitious cells; they view the use of header nodes as little more than old-style hacking. Even so, we will use them here, precisely because they allow us to show the basic pointer manipulations without obscuring the code with special cases. Otherwise, whether or not a header should be used is a matter of personal preference.

As examples, we will write about half of the list ADT routines. First, we need our declarations, which are given in **Figure 3.6.**

```
typedef struct node *node_ptr;
struct node
{
        element_type element;
        node_ptr next;
};
typedef node_ptr LIST;
typedef node_ptr position;
```

Figure 3.6 Type declarations for linked lists

The first function that we will write tests for an empty list. When we write code for any data structure that involves pointers, it is always best to draw a picture first. Figure 3.7 shows an empty list;

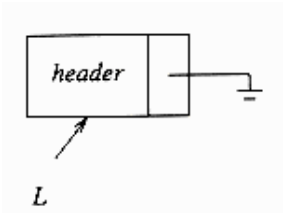


Figure 3.7 Empty list with header

from the figure it is easy to write the function in Figure 3.8.

```
int
is_empty( LIST L )
{
        return( L->next == NULL );
}
```

Figure 3.8

The next function, which is shown in Figure 3.9, tests whether the current element, which by assumption exists, is the last of the list.

```
int
is_last( position p, LIST L )
{
        return( p->next == NULL );
}
```

Figure 3.9 Function to test whether current position is the last in a linked list

The next routine we will write is *find*. *Find*, shown in Figure 3.10, returns the position in the list of some element. Line 2 takes advantage of the fact that the *and* (&&) operation is *short-circuited:* if the first half of the *and* is false, the result is automatically false and the second half is not executed.

```
/* Return position of x in L; NULL if not found */
position
find ( element_type x, LIST L )
{
position p;
/*1*/       p = L->next;
/*2*/       while( (p != NULL) && (p->element != x) )
/*3*/       p = p->next;
/*4*/       return p;
}
```

Figure 3.10 Find routine

Our fourth routine will delete some element *x* in list *L*. We need to decide what to do if *x* occurs more than once or not at all. Our routine deletes the first occurrence of *x* and does nothing if *x* is not in the list. To do this, we find *p*, which is the cell prior to the one containing *x*, via a call to *find_previous*. The code to implement this is shown in Figure 3.11.

```
/* Delete from a list. Cell pointed */
/* to by p->next is wiped out. */
/* Assume that the position is legal. */
/* Assume use of a header node. */
void
delete( element_type x, LIST L )
{
        position p, tmp_cell;
        p = find_previous( x, L );
        if( p->next != NULL )  /* Implicit assumption of header
                                  use */
        {                      /* x is found: delete it */
                tmp_cell = p->next;
                p->next = tmp_cell->next; /* bypass the cell to be
                                             deleted */
                free( tmp_cell );
        }
}
```

Figure 3.11 Deletion routine for linked lists

The *find_previous* routine is similar to *find* and is shown in Figure 3.12.

```
/* Uses a header. If element is not found, then next field */
/* of returned value is NULL */
position
find_previous( element_type x, LIST L )
{
  position p;
  p = L;
  while( (p->next != NULL) && (p->next->element != x) )
       p = p->next;
  return p;
}
```

Figure 3.12 Find_previous--the find routine for use with delete

The last routine we will write is an insertion routine. We will pass an element to be inserted along with the list *L* and a position

p. Our particular insertion routine will insert an element *after* the position implied by *p*. This decision is arbitrary and meant to show that there are no set rules for what insertion does. It is quite possible to insert the new element into position *p* (which means before the element currently in position *p*), but doing this requires knowledge of the element before position *p.* This could be obtained by a call to *find_previous*. It is thus important to comment what you are doing. This has been done in Figure 3.13.

```
/* Insert (after legal position p).*/
/* Header implementation assumed. */
void
insert( element_type x, LIST L, position p )
{
        position tmp_cell;
    tmp_cell = (position) malloc( sizeof (struct node) );
      if( tmp_cell == NULL )

          fatal_error("Out of space!!!");
           else
          {
          tmp_cell->element = x;
          tmp_cell->next = p->next;
          p->next = tmp_cell;
          }
}
```

Figure 3.13 Insertion routine for linked lists

Notice that we have passed the list to the *insert* and *is_last* routines, even though it was never used. We did this because another implementation might need this information, and so not passing the list would defeat the idea of using ADTs.
We could write additional routines to print a list and to perform the *next* function. These are fairly straightforward. We could also write a routine to implement *previous*.

3.2.4. Common Errors

The most common error that you will get is that your program will crash with a nasty error message from the system, such as "memory access violation" or "segmentation violation." This message usually means that a pointer variable contained a bogus address. One common reason is failure to initialize the variable. For

instance, if line (p = L->next;) in Figure 3.14 is omitted, then *p* is undefined and is not likely to be pointing at a valid part of memory. Another typical error would be line (p->next = tmp_cell;) in Figure 3.13. If *p* is *, then the indirection is illegal. This function knows that* p *is not* , so the routine is OK. Of course, you should comment this so that the routine that calls *insert* will insure this. *Whenever you do an indirection, you must make sure that the pointer is not NULL*. Some C compliers will implicity do this check for you, but this is not part of the C standard. When you port a program from one compiler to another, you may find that it no longer works. This is one of the common reasons why.

```
void
delete_list( LIST L )
{
position p;
        p = L->next;        /* header assumed */
        L->next = NULL;
        while( p != NULL )
             {
            free( p );
            p = p->next;
             }
}
```

Figure 3.14 Incorrect way to delete a list

The second common mistake concerns when and when not to use *malloc* to get a new cell. You must remember that declaring a pointer to a structure does not create the structure but only gives enough space to hold the address where some structure might be. The only way to create a record that is not already declared is to use the *malloc* command. The command *malloc*(*size_p*) has the system create, magically, a new structure and return a pointer to it. If, on the other hand, you want to use a pointer variable to run down a list, there is no need to declare a new structure; in that case the *malloc* command is inappropriate. A type cast is used to make both sides of the assignment operator compatible. The C library provides other variations of *malloc* such as *calloc*.

When things are no longer needed, you can issue a *free* command to inform the system that it may reclaim the space. A consequence of the *free(p)* command is that the address that *p* is pointing to is unchanged, but the data that resides at that address is now undefined.

If you never delete from a linked list, the number of calls to *malloc* should equal the size of the list, plus 1 if a header is used. Any less, and you cannot possibly have a working program. Any

more, and you are wasting space and probably time. Occasionally, if your program uses a lot of space, the system may be unable to satisfy your request for a new cell. In this case a *pointer is returned.*

After a deletion in a linked list, it is usually a good idea to free the cell, especially if there are lots of insertions and deletions intermingled and memory might become a problem. You need to keep a temporary variable set to the cell to be disposed of, because after the pointer moves are finished, you will not have a reference to it. As an example, the code in Figure 3.14 is not the correct way to delete an entire list (although it may work on some systems).

Figure 3.15 shows the correct way to do this. One last warning: *malloc*(*sizeof node_ptr*) is legal, but it doesn't allocate enough space for a structure. It allocates space only for a pointer.

```
void
delete_list( LIST L )
{
position p, tmp;
        p = L->next;   /* header assumed */
        L->next = NULL;
        while( p != NULL )
             {
             tmp = p->next;
             free( p );
             p = tmp;
             }
}
```

Figure 3.15 Correct way to delete a list

3.2.5. Doubly Linked Lists

Sometimes it is convenient to traverse lists backwards. The standard implementation does not help here, but the solution is simple. Merely add an extra field to the data structure, containing a pointer to the previous cell. The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand, it simplifies deletion, because you no longer have to refer to a key by using a pointer to the previous cell; this information is now at hand. Figure 3.16 shows a doubly linked list.

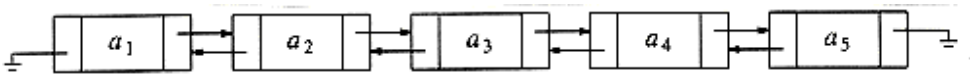


Figure 3.16 A doubly linked list

3.2.5. Doubly Linked Lists

Sometimes it is convenient to traverse lists backwards. The standard implementation does not help here, but the solution is simple. Merely add an extra field to the data structure, containing a pointer to the previous cell. The cost of this is an extra link, which adds to the space requirement and also doubles the cost of insertions and deletions because there are more pointers to fix. On the other hand, it simplifies deletion, because you no longer have to refer to a key by using a pointer to the previous cell; this information is now at hand. Figure 3.16 shows a doubly linked list.

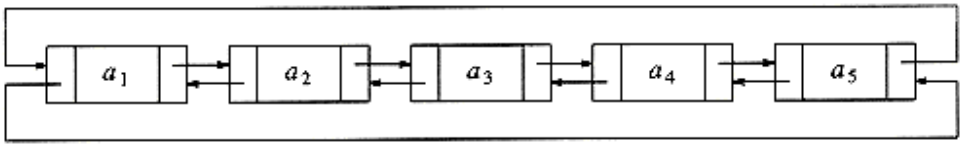


Figure 3.17 A double circularly linked list

3.2.5. Multi-Linked Lists

In a general multi-linked list each node can have any number of pointers to other nodes, and there may or may not be inverses for each pointer.

A university with 40,000 students and 2,500 courses needs to be able to generate two types of reports. The first report lists the class registration for each class, and the second report lists, by student, the classes that each student is registered for.

The obvious implementation might be to use a two-dimensional array. Such an array would have 100 million entries. The average student registers for about three courses, so only 120,000 of these entries, or roughly 0.1 percent, would actually have meaningful data.

What is needed is a list for each class, which contains the students in the class. We also need a list for each student, which contains the classes the student is registered for. Figure 3.18 shows our implementation.

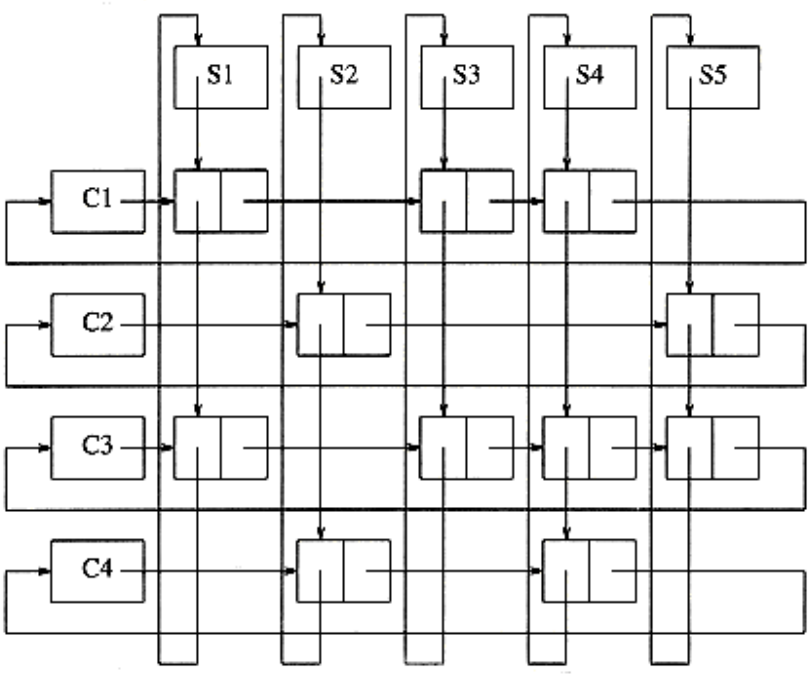


Figure 3.18 Multilinkedlist implementation for registration problem

As the figure shows, we have combined two lists into one. All lists use a header and are circular. To list all of the students in class C3, we start at C3 and traverse its list (by going right). The first cell belongs to student S1. Although there is no explicit information to this effect, this can be determined by following the student's linked list until the header is reached. Once this is done, we return to C3's list (we stored the position we were at in the course list before we traversed the student's list) and find another cell, which can be determined to belong to S3. We can continue and find that S4 and S5 are also in this class. In a similar manner, we can determine, for any student, all of the classes in which the student is registered.

Using a circular list saves space but does so at the expense of time. In the worst case, if the first student was registered for every course, then every entry would need to be examined in order to determine all the course names for that student. Because in this application there are relatively few courses per student and few students per course, this is not likely to happen. If it were suspected that this could cause a problem, then each of the (nonheader) cells could have pointers directly back to the student and class header. This would double the space requirement, but simplify and speed up the implementation.

Check your Progress

1. Write a program to print the elements of a singly linked list.
2. Write a program to swap two adjacent elements by adjusting only the pointers (and not the data) using
    a. singly linked lists,
    b. doubly linked lists
3. Write a function to add two polynomials. Do not destroy the input. Use a linked list implementation. If the polynomials have *m* and *n* terms respectively.

3.3. THE STACK ADT

A stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.

A stack is a *restricted data structure*, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest.

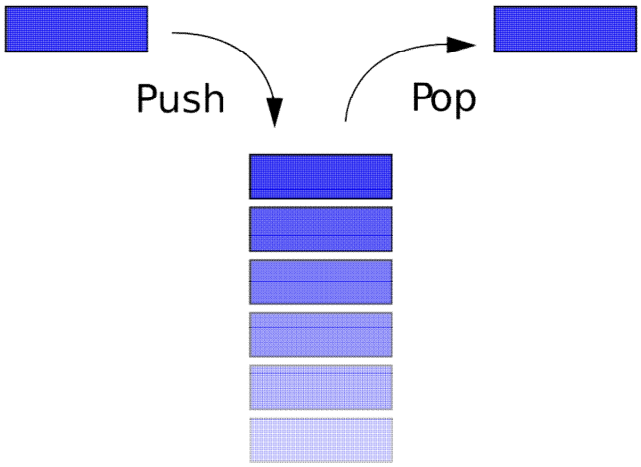


Figure 3.19 Simple representation of stack

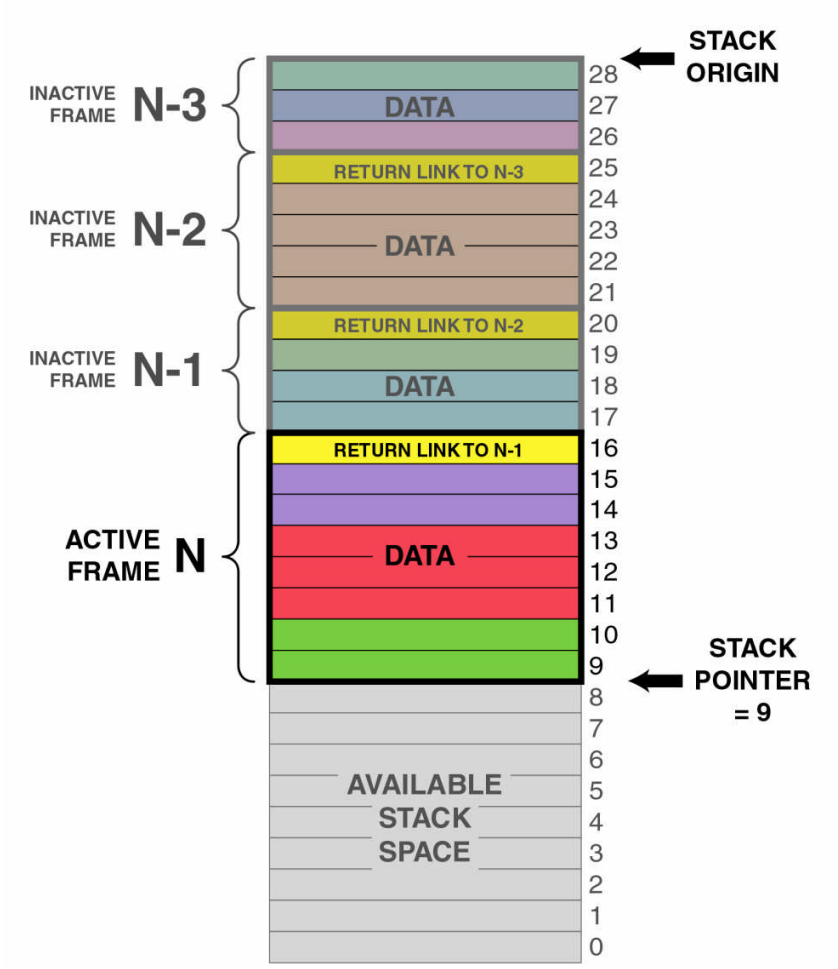


Figure 3.20 Basic Architecture of stack

3.3.1. Stack Model

A *stack* is a list with the restriction that *inserts* and *deletes* can be performed in only one position, namely the end of the list called the *top*. The fundamental operations on a stack are *push*, which is equivalent to an insert, and *pop*, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a *pop* by use of the *top* routine. A *pop* or *top* on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a *push* is an implementation error but not an ADT error.

Stacks are sometimes known as LIFO (last in, first out) lists. The model depicted in Figure 3.21 signifies only that *pushes* are input operations and *pops* and *tops* are output. The usual operations to make empty stacks and test for emptiness are part of the repertoire, but essentially all that you can do to a stack is *push* and *pop*.

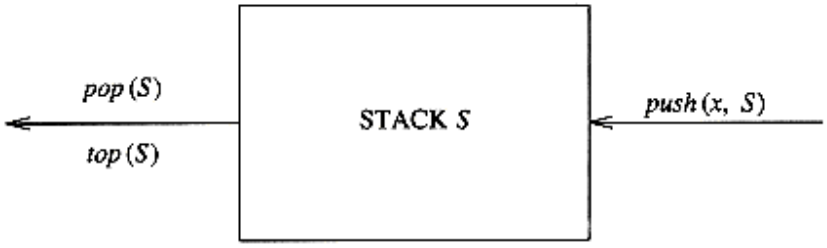


Figure 3.21 Stack model: input to a stack is by push, output is by pop

Figure 3.22 shows an abstract stack after several operations. The general model is that there is some element that is at the top of the stack, and it is the only element that is visible.

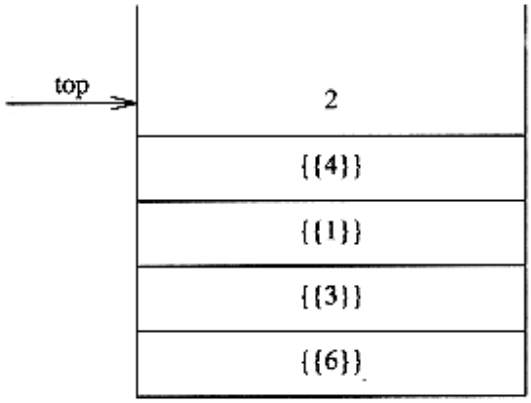


Figure 3.22 Stack model: only the top element is accessible

3.3.2. Implementation of Stacks

Two implementation of stacks are discussed here. Linked list implementation and array implementation.

3.3.2.1 Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a *push* by inserting at the front of the list. We perform a *pop* by deleting the element at the front of the list. A *top* operation merely examines the element at the front of the list, returning its value. Sometimes the *pop* and *top* operations are combined into one. We could use calls to the linked list routines of the previous section, but we will rewrite the stack routines from scratch for the sake of clarity.

First, we give the definitions in Figure 3.23. We implement the stack using a header.

```
typedef struct node *node_ptr;
struct node
{
        element_type element;
        node_ptr next;
};
typedef node_ptr STACK;
```

Figure 3.23 Type declaration for linked list implementation of the stack ADT

Then Figure 3.24 shows that an empty stack is tested for in the same manner as an empty list.

```
int
is_empty( STACK S )
{
        return( S->next == NULL );
}
```

Figure 3.24 Routine to test whether a stack is empty-linked list implementation

Creating an empty stack is also simple. We merely create a header node; *make_null* sets the *next* pointer to *NULL* (see Fig. 3.25).

```
STACK
create_stack( void )
{
        STACK S;
        S = (STACK) malloc( sizeof( struct node ) );
        if( S == NULL )
                fatal_error("Out of space!!!");
        return S;
}
void
make_null( STACK S )
{
        if( S != NULL )
                S->next = NULL;
        else
                error("Must use create_stack first");
}
```

Figure 3.25 Routine to create an empty stack-linked list implementation

The *push* is implemented as an insertion into the front of a linked list, where the front of the list serves as the top of the stack (see Fig. 3.26).

```
void
push( element_type x, STACK S )
{
        node_ptr tmp_cell;
        tmp_cell = (node_ptr) malloc( sizeof ( struct node ) );
        if( tmp_cell == NULL )
                fatal_error("Out of space!!!");
        else
        {
                tmp_cell->element = x;
                tmp_cell->next = S->next;
                S->next = tmp_cell;
        }
}
```

Figure 3.26 Routine to push onto a stack-linked list implementation

The *top* is performed by examining the element in the first position of the list (see Fig. 3.27).

```
element_type
top( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->next->element;
}
```

Figure 3.27 Routine to return top element in a stack--linked list implementation

Finally, we implement *pop* as a delete from the front of the list (see Fig. 3.28).

```
void
pop( STACK S )
{
        node_ptr first_cell;
        if( is_empty( S ) )
                error("Empty stack");
        else
        {
                first_cell = S->next;
                S->next = S->next->next;
                free( first_cell );
        }
}
```

Figure 3.28 Routine to pop from a stack--linked list implementation

The drawback of this implementation is that the calls to *malloc* and *free* are expensive, especially in comparison to the pointer manipulation routines. Some of this can be avoided by using a second stack, which is initially empty. When a cell is to be disposed from the first stack, it is merely placed on the second stack. Then, when new cells are needed for the first stack, the second stack is checked first.

One problem that affects the efficiency of implementing stacks is error testing. Our linked list implementation carefully checked for errors. A *pop* on an empty stack or a *push* on a full stack will overflow the array bounds and cause a crash. This is undesirable, but if checks for these conditions were put in the array implementation, they would likely take as much time as the actual stack manipulation. For this reason, it has become a common practice to skip on error checking in the stack routines, except where error handling is crucial (as in operating systems). Although you can probably get away with this in most cases by declaring the stack to be large enough not to overflow and ensuring that routines that use *pop* never attempt to *pop* an empty stack, this can lead to code that barely works at best, especially when programs get large and are written by more than one person or at more than one time. Because stack operations take such fast constant time, it is rare that a significant part of the running time of a program is spent in these routines. This means that it is generally not justifiable to omit error checks. You should always write the error checks; if they are redundant, you can always comment them out if they really cost too much time.

3.3.2.1 Array implementation of Stacks

An alternative implementation avoids pointers and is probably the more popular solution. The only potential hazard with this strategy is that we need to declare an array size ahead of time. Generally this is not a problem, because in typical applications, even if there are quite a few stack operations, the actual number of elements in the stack at any time never gets too large. It is usually easy to declare the array to be large enough without wasting too much space. If this is not possible, then a safe course would be to use a linked list implementation.

If we use an array implementation, the implementation is trivial. Associated with each stack is the top of stack, *tos*, which is -1 for an empty stack (this is how an empty stack is initialized). To push some element *x* onto the stack, we increment *tos* and then set *STACK*[*tos*] = *x*, where *STACK* is the array representing the actual stack. To pop, we set the return value to *STACK*[*tos*] and then decrement *tos*. Of course, since there are potentially several stacks, the *STACK* array and *tos* are part of one structure representing a stack. It is almost always a bad idea to use global variables and fixed names to represent this (or any) data structure, because in most real-life situations there will be more than one stack. When writing your actual code, you should attempt to follow the model as closely as possible, so that no part of your code, except for the stack routines, can attempt to access the array or top-of-stack variable implied by each stack. This is true for *all* ADT operations.

A *STACK* is defined in Figure 3.29 as a pointer to a structure. The structure contains the *top_of_stack* and *stack_size* fields.

```
struct stack_record
{
        unsigned int stack_size;
        int top_of_stack;
        element_type *stack_array;
};
typedef struct stack_record *STACK;
#define EMPTY_TOS (-1) /* Signifies an empty stack */
```

Figure 3.29 STACK definition--array implementaion

Once the maximum size is known, the stack array can be dynamically allocated. Figure 3.30 creates a stack of a given maximum size. Lines 3-5 allocate the stack structure, and lines 6-8 allocate the stack array. Lines 9 and 10 initialize the *top_of_stack* and *stack_size* fields. The stack array does not need to be initialized. The stack is returned at line 11.

```
STACK
create_stack( unsigned int max_elements )
{
STACK S;
/*1*/       if( max_elements < MIN_STACK_SIZE )
/*2*/           error("Stack size is too small");
/*3*/       S = (STACK) malloc(sizeof(struct stack_record) );
/*4*/       if( S == NULL )
/*5*/           fatal_error("Out of space!!!");
/*6*/       S->stack_array = (element_type *)
                        malloc( sizeof( element_type ) * max_elements );
/*7*/       if( S->stack_array == NULL )
/*8*/           fatal_error("Out of space!!!");
/*9*/       S->top_of_stack = EMPTY_TOS;
/*10*/      S->stack_size = max_elements;
/*11*/      return( S );
}
```

Figure 3.30 Stack creation--array implementaion

The routine *dispose_stack* should be written to free the stack structure. This routine first frees the stack array and then the stack structure (See Figure 3.31). Since *create_stack* requires an argument in the array implementation, but not in the linked list implementation, the routine that uses a stack will need to know which implementation is being used unless a dummy parameter is added for the later implementation. Unfortunately, efficiency and software idealism often create conflicts.

```
void
dispose_stack( STACK S )
{
        if( S != NULL )
        {
                free( S->stack_array );
                free( S );
        }
}
```

Figure 3.31 Routine for freeing stack--array implementation

We have assumed that all stacks deal with the same type of element.

We will now rewrite the four stack routines. In true ADT spirit, we will make the function and procedure heading look

identical to the linked list implementation. The routines themselves are very simple and follow the written description exactly (see Figs. 3.32 to 3.36).

Pop is occasionally written as a function that returns the popped element (and alters the stack). Although current thinking suggests that functions should not change their input variables, Figure 3.37 illustrates that this is the most convenient method in C.

```
int
is_empty( STACK S )
{
        return( S->top_of_stack == EMPTY_TOS );
}
```

Figure 3.32 Routine to test whether a stack is empty--array implementation

```
void
make_null( STACK S )
{
        S->top_of_stack = EMPTY_TOS;
}
```

Figure 3.33 Routine to create an empty stack--array implementation

```
void
push( element_type x, STACK S )
{
        if( is_full( S ) )
                error("Full stack");
        else
                S->stack_array[ ++S->top_of_stack ] = x;
}
```

Figure 3.34 Routine to push onto a stack--array implementation

```
element_type
top( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->stack_array[ S->top_of_stack ];
}
```

Figure 3.35 Routine to return top of stack--array implementation

```
void
pop( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                S->top_of_stack--;
}
```

Figure 3.36 Routine to pop from a stack--array implementation

```
element_type
pop( STACK S )
{
        if( is_empty( S ) )
                error("Empty stack");
        else
                return S->stack_array[ S->top_of_stack-- ];
}
```

Figure 3.37 Routine to give top element and pop a stack--array implementation

Check your Progress

1. Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.
2. Propose a data structure that supports the stack push and pop operations and a third operation find_min, which returns the smallest element in the data structure, all in O(1) worst case time.
3. Show how to implement three stacks in one array.

3.4. THE QUEUE ADT

A *queue* is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that whenever an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

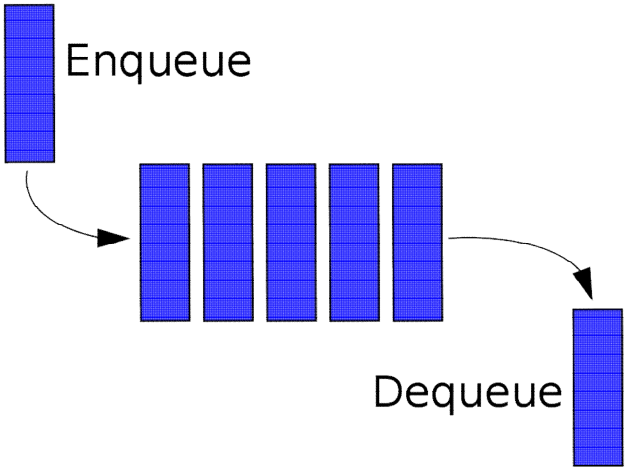


Fig. 3.38 Simple representation of a queue

Like stacks, *queues* are lists. With a queue, however, insertion is done at one end, whereas deletion is performed at the other end.

Queues provide services in computer science, transport and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer.

Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists.

3.4.1. Queue Model

The basic operations on a queue are *enqueue,* which inserts an element at the end of the list (called the rear), and *dequeue,* which deletes (and returns) the element at the start of the list (known as the front). Figure 3.39 shows the abstract model of a queue.

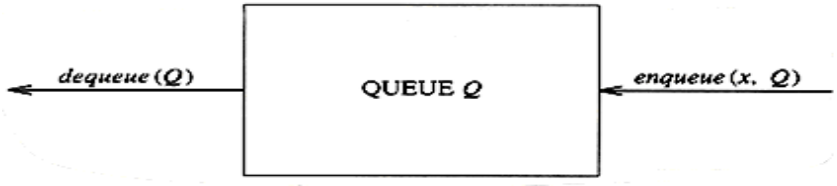


Figure 3.39 Model of a queue

3.3.2. Implementation of Queues

Two implementation of queue are discussed here. Linked list implementation and array implementation.

3.3.2.1 Linked List implementation

```
typedef struct list
{
    int data;
    struct list *next;
}node;
```

Figure 3.40 Definitions for Linked list implementation of queue

```
node *insert(node **f,node **r)
{
    node *ln;
    ln=(node *)malloc(sizeof(node));
    printf("enter the value:");
    scanf("%d",&ln->data);
    ln->next=NULL;
    (*r)->next=(node *)malloc(sizeof(node));
    if(*f==NULL)
    {
        *f=ln;
        *r=ln;
    }
    else
    {
        (*r)->next=ln;
        *r=ln;
    }
}
```

Figure 3.41 Inserting in a queue

```
display(node *ln)
{
    node *tmp;
    while(ln!=NULL)
    {
        printf("%d->",ln->data);
        ln=ln->next;
    }
}
```

Figure 3.41 Displaying a queue

```
delete(node **f,node **r)
{
    node *tmp;
    tmp=*f;
    if(*f==NULL)
    {
        printf("queue empty");
    }
    else
    {
        (*f)=(*f)->next;
        free(tmp);
    }
}
```

Figure 3.42 Deleting from a queue

3.3.2.2 Array Implementation of Queues

For each queue data structure, we keep an array, *QUEUE*[], and the positions *q_front* and *q_rear,* which represent the ends of the queue. We also keep track of the number of elements that are actually in the queue, *q_size*. All this information is part of one structure, and as usual, except for the queue routines themselves, no routine should ever access these directly. The following figure shows a queue in some intermediate state. By the way, the cells that are blanks have undefined values in them. In particular, the first two cells have elements that used to be in the queue.

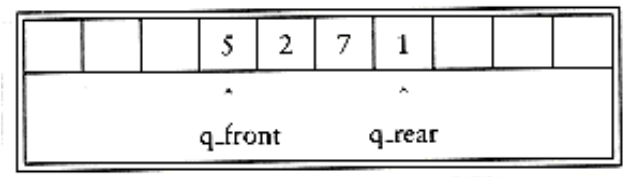


Figure 3.43 Sample queue

The operations should be clear. To *enqueue* an element *x*, we increment *q_size* and *q_rear*, then set *QUEUE*[*q_rear*] = *x*. To *dequeue* an element, we set the return value to *QUEUE*[*q_front*], decrement *q_size*, and then increment *q_front*. Other strategies are possible (this is discussed later). We will comment on checking for errors presently.

There is one potential problem with this implementation. After 10 enqueues, the queue appears to be full, since *q_front* is now 10, and the next *enqueue* would be in a nonexistent position. However, there might only be a few elements in the queue, because several elements may have already been dequeued. Queues, like stacks, frequently stay small even in the presence of a lot of operations.

The simple solution is that whenever *q_front* or *q_rear* gets to the end of the array, it is wrapped around to the beginning. The following figure shows the queue during some operations. This is known as a *circular array* implementation.

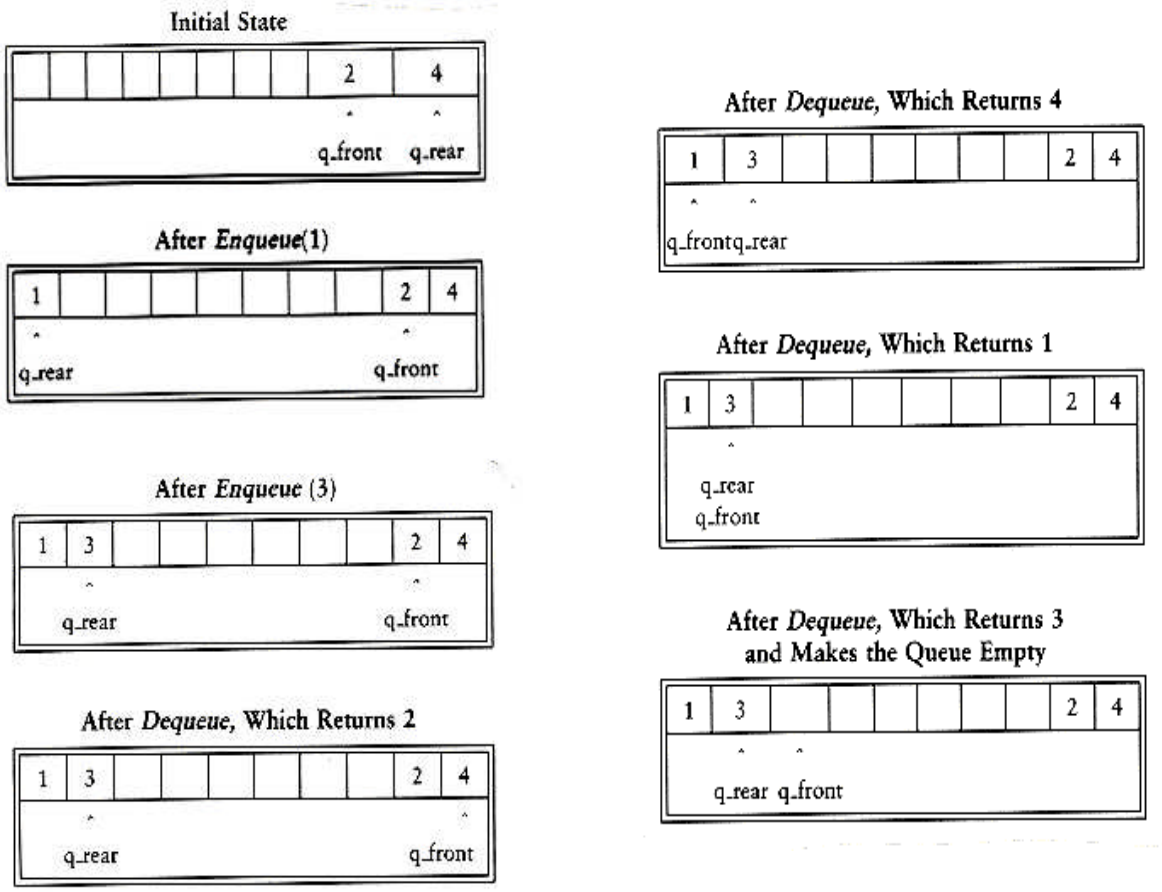


Figure 3.44 Circular array implementation of queue

```
struct queue_record
{
        unsigned int q_max_size;  /* Maximum # of elements */
        /* until Q is full */
        unsigned int q_front;
        unsigned int q_rear;
        unsigned int q_size;      /* Current # of elements in Q */
        element_type *q_array;
};
typedef struct queue_record * QUEUE;
```

Figure 3.45 Type declarations for queue--array implementation

```
int
is_empty( QUEUE Q )
{
        return( Q->q_size == 0 );
}
```

Figure 3.45 Routine to test whether a queue is empty-array implementation

```
void
make_null ( QUEUE Q )
{
        Q->q_size = 0;
        Q->q_front = 1;
        Q->q_rear = 0;
}
```

Figure 3.46 Routine to make an empty queue-array implementation

```
unsigned int
succ( unsigned int value, QUEUE Q )
{
        if( ++value == Q->q_max_size )
                value = 0;
        return value;
}
void
enqueue( element_type x, QUEUE Q )
{
        if( is_full( Q ) )
                error("Full queue");
        else
        {
                Q->q_size++;
                Q->q_rear = succ( Q->q_rear, Q );
                Q->q_array[ Q->q_rear ] = x;
        }
}
```

Figure 3.47 Routine to enqueue-array implementation

Check your Progress

1. Write the routines to implement queues using
    1. linked lists
    2. arrays
2. A *deque* is a data structure consisting of a list of items, on which the following operations are possible:

    push(x,d): Insert item *x* on the front end of deque *d.*

    pop(d): Remove the front item from deque *d* and return it.

    inject(x,d): Insert item *x* on the rear end of deque *d.*

    eject(d): Remove the rear item from deque *d* and return it.
    Write routines to support the deque that take $O(1)$ time per operation.

Let us sum up

This chapter describes the concept of ADTs and illustrates the concept with three of the most common abstract data types. The primary objective is to separate the implementation of the abstract data types from their function. The program must know what the operations do, but it is actually better off not knowing how it is done.

Lists, stacks, and queues are perhaps the three fundamental data structures in all of computer science, and their use is documented through a host of examples. In particular, we saw how stacks are used to keep track of procedure and function calls and how recursion is actually implemented. This is important to understand, not just because it makes procedural languages possible, but because knowing how recursion is implemented removes a good deal of the mystery that surrounds its use. Although recursion is very powerful, it is not an entirely free operation; misuse and abuse of recursion can result in programs crashing.

References:

1. Data structure – A Pseudocode Approach with C – Richard F Gilberg Behrouz A. Forouzan, Thomson
2. Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2nd Edition
3. Data structures & Program Design in C Robert Kruse, C. L.Tondo, Bruce Leung Pearson
4. "Data structure using C" AM Tanenbaum, Y Langsam & M J Augustein, Prentice Hall India

Question Pattern:

1. What are abstract data types? Explain.
2. Explain the concept of stack in detail.
3. Explain the concept of queue in detail.

4

INTRODUCTION TO TREES

Unit Structure:

4.1 Trees
4.2 Binary Trees
4.3 Expression Trees
4.4 AVL Trees
4.5 Splay Trees
4.6 The Search Tree ADT-Binary Search Trees

4.1 TREES

A *tree* can be defined in several ways. One way to define a tree is recursively. A tree is a collection of nodes. The collection can be empty, which is sometimes denoted as A. Otherwise, a tree consists of a distinguished node *r*, called the *root*, and zero or more (sub)trees T_1, T_2, . . . , T_k, each of whose roots are connected by a directed *edge* to *r*.

The root of each subtree is said to be a *child* of *r*, and *r* is the *parent* of each subtree root. Figure 4.1 shows a typical tree using the recursive definition.

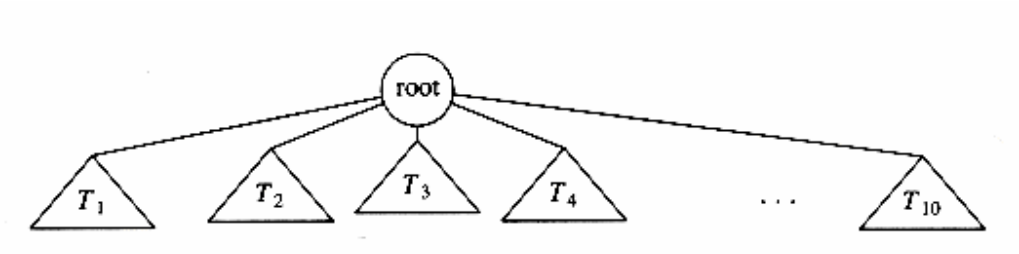


Figure 4.1 Generic tree

From the recursive definition, we find that a tree is a collection of *n* nodes, one of which is the root, and *n* - 1 edges. Having *n* - 1 edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent (see Fig. 4.2).

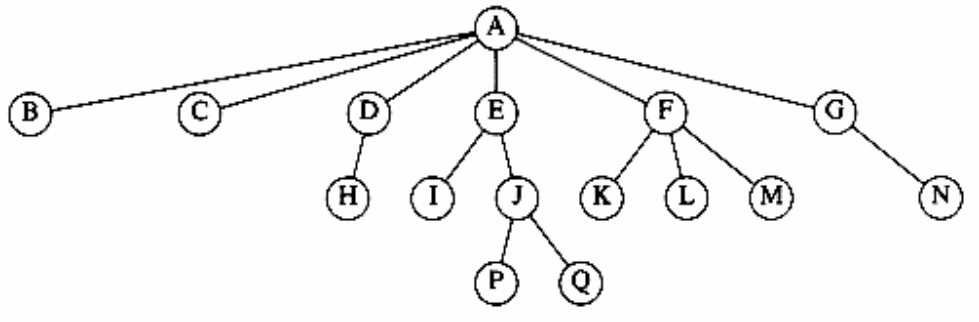


Figure 4.2 A tree

In the tree of Figure 4.2, the root is *A*. Node *F* has *A* as a parent and *K*, *L*, and *M* as children. Each node may have an arbitrary number of children, possibly zero. Nodes with no children are known as *leaves*; the leaves in the tree above are *B*, *C*, *H*, *I*, *P*, *Q*, *K*, *L*, *M*, and *N*. Nodes with the same parent are *siblings*; thus *K*, L, and *M* are all siblings. *Grandparent* and *grandchild* relations can be defined in a similar manner.

A *path* from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 < i < k$. The *length* of this path is the number of edges on the path, namely k -1. There is a path of length zero from every node to itself. Notice that in a tree there is exactly one path from the root to each node.

For any node n_i, the *depth* of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0. The *height* of n_i is the longest path from n_i to a leaf. Thus all leaves are at height 0. The height of a tree is equal to the height of the root. For the tree in Figure 4.2, *E* is at depth 1 and height 2; *F* is at depth 1 and height 1; the height of the tree is 3. The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.

If there is a path from n_1 to n_2, then n_1 is an *ancestor* of n_2 and n_2 is a *descendant* of n_1.

4.1.1. Implementation of Trees

One way to implement a tree would be to have in each node, besides its data, a pointer to each child of the node. However, since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the children direct links in the data structure, because there would be too much wasted space. The solution is simple: Keep the children of each node in a linked list of tree nodes. The declaration in Figure 4.3 is typical.

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
        element_type element;
        tree_ptr first_child;
        tree_ptr next_sibling;
};
```

Figure 4.3 Node declarations for trees

Figure 4.4 shows how a tree might be represented in this implementation. Arrows that point downward are *first_child* pointers. Arrows that go left to right are *next_sibling* pointers. Null pointers are not drawn, because there are too many.

In the tree of Figure 4.4, node *E* has both a pointer to a sibling (*F*) and a pointer to a child (*I*), while some nodes have neither.

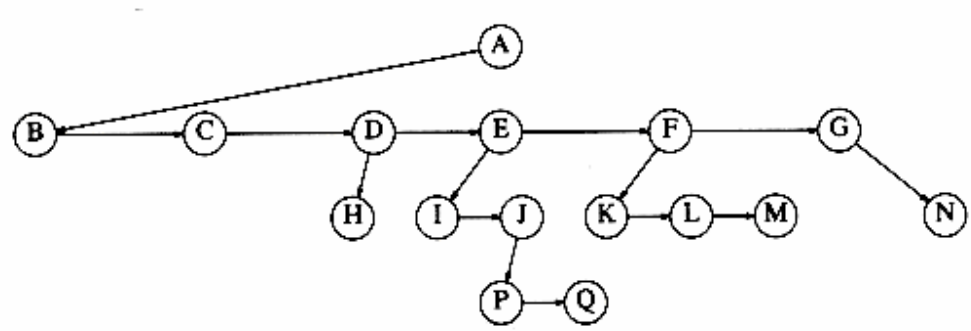
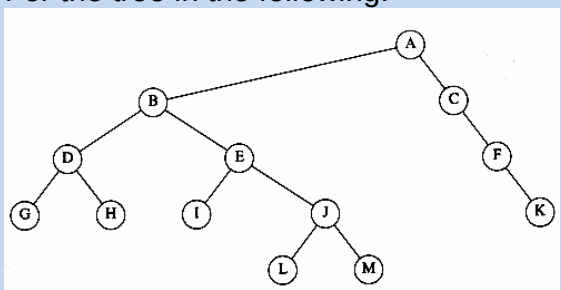


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Check your progress

1. For the tree in the following:



   a. Which node is the root?
   b. Which nodes are leaves?

2. For each node in the tree of Figure above:

   a. *Name the parent node.*
   b. *List the children.*
   c. *List the siblings.*
   d. *Compute the depth.*
   e. *Compute the height*.

4.2 BINARY TREES:

A binary tree is a tree in which no node can have more than two children.

Figure 4.5 shows that a binary tree consists of a root and two subtrees, T_l and T_r, both of which could possibly be empty

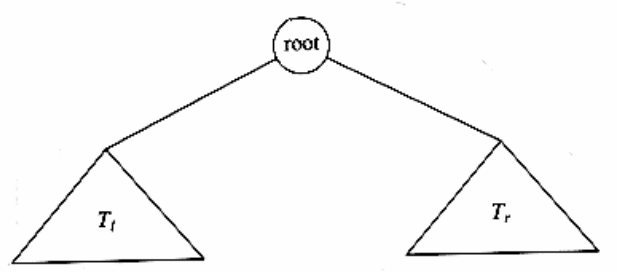


Figure 4.5 Generic binary tree

A property of a binary tree that is sometimes important is that the depth of an average binary tree is considerably smaller than *n*. Unfortunately, the depth can be as large as *n* -1, as the example in Figure 4.6 shows.

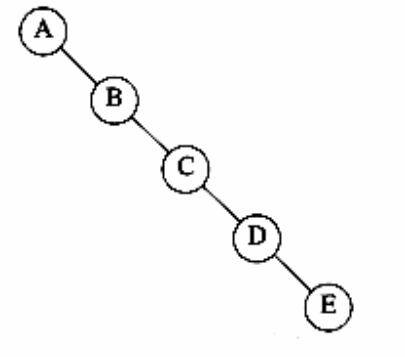


Figure 4.6 Worst-case binary tree

4.2.1 Implementation

Because a binary tree has at most two children, we can keep direct pointers to them. The declaration of tree nodes is similar in structure to that for doubly linked lists, in that a node is a structure consisting of the *key* information plus two pointers (*left* and *right*) to other nodes.

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
        element_type element;
        tree_ptr left;
        tree_ptr right;
};
typedef tree_ptr TREE;
```

Figure 4.7 Binary tree node declarations

Many of the rules that apply to linked lists will apply to trees as well. In particular, when an insertion is performed, a node will have to be created by a call to *malloc*. Nodes can be freed after deletion by calling *free*.

We could draw the binary trees using the rectangular boxes that are customary for linked lists, but trees are generally drawn as circles connected by lines, because they are actually graphs. We also do not explicitly draw *NULL* pointers when referring to trees, because every binary tree with *n* nodes would require *n* + 1 *NULL* pointers.

Binary trees have many important uses not associated with searching. One of the principal uses of binary trees is in the area of compiler design.

4.2.2 Types of binary trees:

- A **rooted binary tree** is a rooted tree in which every node has at most two children.
- A **full binary tree** (sometimes **proper binary tree** or **2-tree** or **strictly binary tree**) is a tree in which every node other than the leaves has two children.
- A **perfect binary tree** is a full binary tree in which all leaves are at the same depth or same level. (This is ambiguously also called a **complete binary tree**.)
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- An **infinite complete binary** tree is a tree with N_0 levels, where for each level d the number of existing nodes at level d is equal to 2d. The cardinal number of the set of all nodes is N_0. The cardinal number of the set of all paths is 2^{N_0}.
- A **balanced binary tree** is where the depth of all the leaves differs by at most 1. Balanced trees have a predictable depth (how many nodes are traversed from the root to a leaf, root counting as node 0 and subsequent as 1, 2, ..., depth). This depth is equal to the integer part of $log_2(n)$ where n is the number of nodes on the balanced tree. Example 1: balanced tree with 1 node, $log_2(1)$ = 0 (depth = 0). Example 2: balanced tree with 3 nodes, $log_2(3)$ = 1.59 (depth=1). Example 3: balanced tree with 5 nodes, $log_2(5)$ = 2.32 (depth of tree is 2 nodes).
- A **rooted complete binary tree** can be identified with a free magma.
- A **degenerate tree** is a tree where for each parent node, there is only one associated child node. This means that in a performance measurement, the tree will behave like a linked list data structure.

A rooted tree has a top node as root.

4.2.3 Properties of binary trees

- The number of nodes *n* in a perfect binary tree can be found using this formula: $n = 2^{h+1} - 1$ where *h* is the height of the tree.
- The number of nodes *n* in a complete binary tree is minimum: $n = 2^h$ and maximum: $n = 2^{h+1} - 1$ where *h* is the height of the tree.
- The number of nodes *n* in a perfect binary tree can also be found using this formula: $n = 2L - 1$ where *L* is the number of leaf nodes in the tree.
- The number of leaf nodes *n* in a perfect binary tree can be found using this formula: $n = 2^h$ where *h* is the height of the tree.
- The number of NULL links in a Complete Binary Tree of n-node is (n+1).
- The number of leaf node in a Complete Binary Tree of n-node is *UpperBound* (*n* / 2).
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$.

4.2.4 Tree Traversals

In computer science, **tree-traversal** refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Compared to linear data structures like linked lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node. Thus the process is most easily described through recursion.

To traverse a non-empty binary tree in **preorder**, perform the following operations recursively at each node, starting with the root node:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

(This is also called Depth-first traversal.)

To traverse a non-empty binary tree in **inorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

(This is also called **Symmetric traversal**.)

To traverse a non-empty binary tree in **postorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Finally, trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level. This is also called Breadth-first traversal.

Example

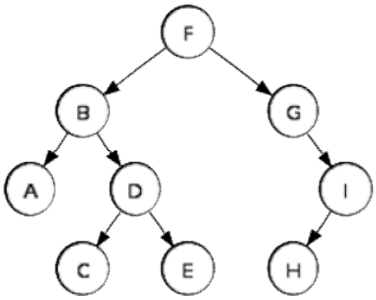


Figure 4.8 Binary tree

In this binary tree fig. 4.8

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

4.2.5 Sample implementations

```
preorder(node)
  print node.value
  if node.left ≠ null then preorder(node.left)
  if node.right ≠ null then preorder(node.right)
```

Figure 4.9 Sample preorder implementation

```
inorder(node)
  if node.left  ≠ null then inorder(node.left)
  print node.value
  if node.right ≠ null then inorder(node.right)
```

Figure 4.10 Sample inorder implementation

```
postorder(node)
  if node.left  ≠ null then postorder(node.left)
  if node.right ≠ null then postorder(node.right)
  print node.value
```

Figure 4.11 Sample inorder implementation

All sample implementations will require call stack space proportional to the height of the tree. In a poorly balanced tree, this can be quite considerable.

4.3 EXPRESSION TREES:

Figure 4.12 shows an example of an *expression tree*. The leaves of an expression tree are *operands*, such as constants or variable names, and the other nodes contain *operators*. This particular tree happens to be binary, because all of the operations are binary, and although this is the simplest case, it is possible for nodes to have more than two children. It is also possible for a node to have only one child, as is the case with the *unary minus* operator. We can evaluate an expression tree, *T*, by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees. In our example, the left subtree evaluates to *a* + (*b* * *c*) and the right subtree evaluates to ((*d* **e*) + *f*)**g.* The entire tree therefore represents (*a* + (*b***c*)) + (((*d* * *e*) + *f*)* *g*).

We can produce an (overly parenthesized) infix expression by recursively producing a parenthesized left expression, then printing out the operator at the root, and finally recursively producing a parenthesized right expression. This general strattegy (left, node, right) is known as an *inorder* traversal; it is easy to remember because of the type of expression it produces.

An alternate traversal strategy is to recursively print out the left subtree, the right subtree, and then the operator. If we apply this strategy to our tree above, the output is a b c * + d e * f + g * +, which is easily seen to be the postfix representation. This traversal strategy is generally known as a *postorder* traversal.

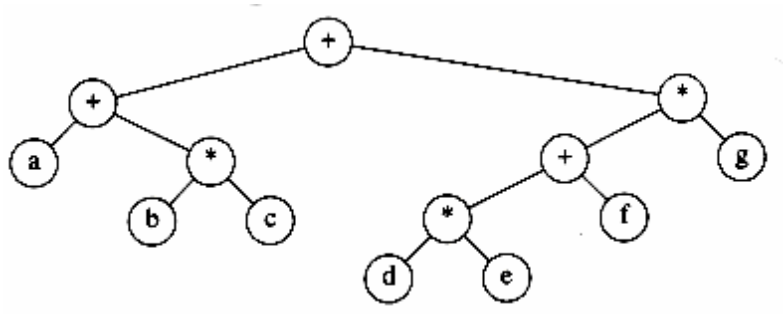


Figure 4.12 Expression tree for (a + b * c) + ((d * e + f) * g)

A third traversal strategy is to print out the operator first and then recursively print out the left and right subtrees. The resulting expression, + + a * b c * + * d e f g, is the less useful *prefix* notation and the traversal strategy is a *preorder* traversal.

4.3.1 Constructing an Expression Tree

We now give an algorithm to convert a postfix expression into an expression tree. Since we already have an algorithm to convert infix to postfix, we can generate expression trees from the two common types of input. We read our expression one symbol at a time. If the symbol is an operand, we create a one-node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

As an example, suppose the input is a b + c d e + * *
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*

*For convenience, we will have the stack grow from left to right in the diagrams.

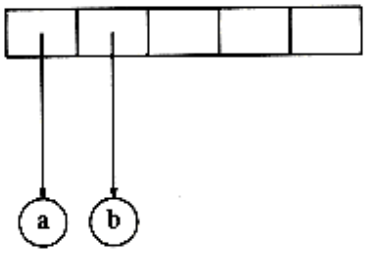


Figure 4.13 Pushing pointers to a and b to stack

Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

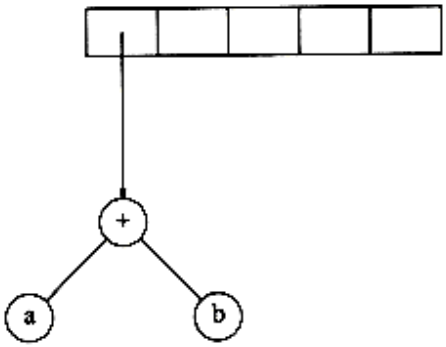


Figure 4.14 Pushing pointers to + to stack

Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

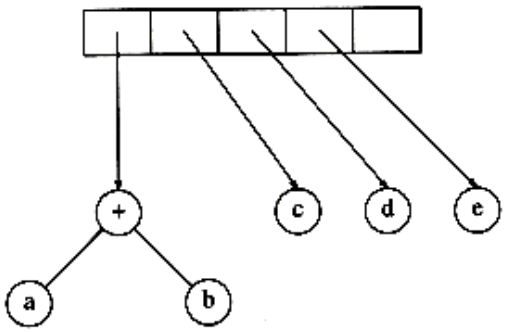


Figure 4.15 Pushing pointers to c, d and e to stack

Now a '+' is read, so two trees are merged.

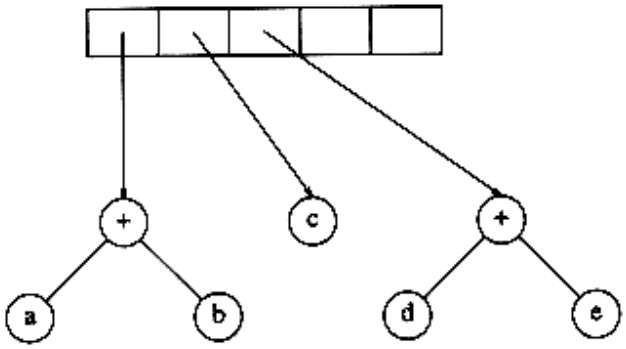


Figure 4.16 Merging two trees

Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.

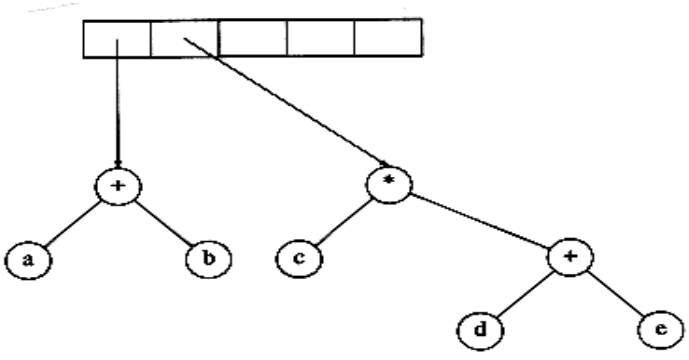


Figure 4.17 Popping two tree pointers and forming a new tree with a '*' as root

Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.

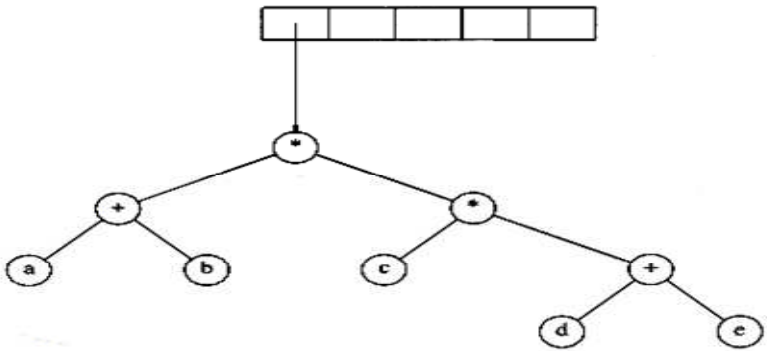
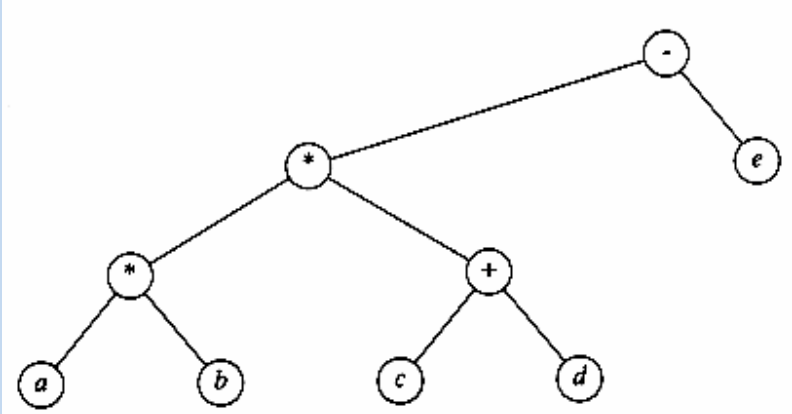


Figure 4.18 Final tree

Check your progress:

1. Give the prefix, infix, and postfix expressions corresponding to the tree in the following Figure:



2. Convert the infix expression a + b * c + (d * e + f) * g into postfix.

4.4. AVL TREES:

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a *balance* condition. The balance condition must be easy to maintain, and it ensures that the depth of the tree is $O(\log n)$. The simplest idea is to require that the left and right subtrees have the same height. As Figure 4.19 shows, this idea does not force the tree to be shallow.

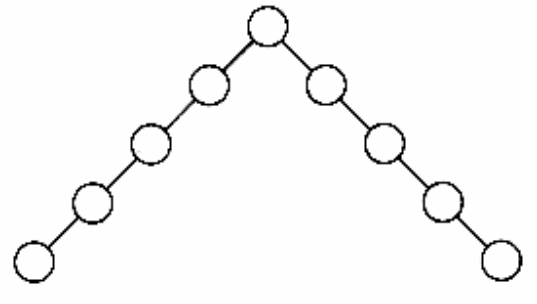


Figure 4.19 A bad binary tree. Requiring balance at the root is not enough.

Another balance condition would insist that every node must have left and right subtrees of the same height. If the height of an empty subtree is defined to be -1 (as is usual), then only perfectly balanced trees of 2^k - 1 nodes would satisfy this criterion. Thus, although this guarantees trees of small depth, the balance condition is too rigid to be useful and needs to be relaxed.

An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.) In Figure 4.20 the tree on the left is an AVL tree, but the tree on the right is not. Height information is kept for each node (in the node structure). It is easy to show that the height of an AVL tree is at most roughly 1.44 log(n + 2) - .328, but in practice it is about log(n + 1) + 0.25 (although the latter claim has not been proven). As an example, the AVL tree of height 9 with the fewest nodes (143) is shown in Figure 4.21. This tree has as a left subtree an AVL tree of height 7 of minimum size. The right subtree is an AVL tree of height 8 of minimum size. This tells us that the minimum number of nodes, $N(h)$, in an AVL tree of height h is given by $N(h) = N(h-1) + N(h-2) + 1$. For $h = 0$, $N(h) = 1$. For $h = 1$, $N(h) = 2$. The function $N(h)$ is closely related to the Fibonacci numbers, from which the bound claimed above on the height of an AVL tree follows.

Thus, all the tree operations can be performed in $O(\log n)$ time, except possibly insertion (we will assume lazy deletion). When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is potentially difficult is that inserting a node could violate the AVL tree property. (For instance, inserting 6½ into the AVL tree in Figure 4.20 would destroy the balance condition at the node with key 8.) If this is the case, then the property has to be restored before the insertion step is considered over. It turns out that this can always be done with a simple modification to the tree, known as a rotation.

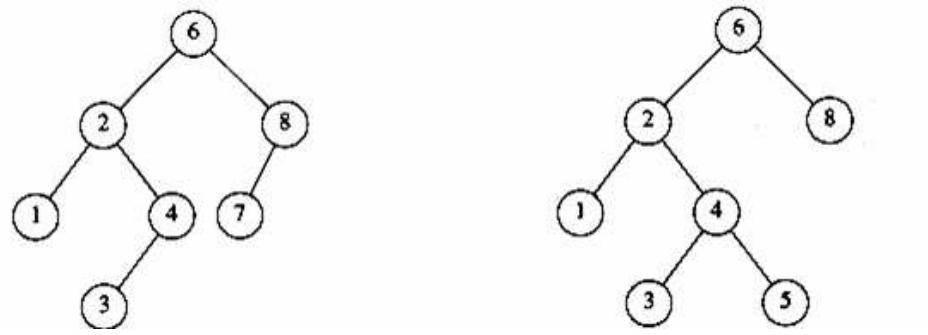


Figure 4.20 Two binary search trees. Only the left tree is AVL.

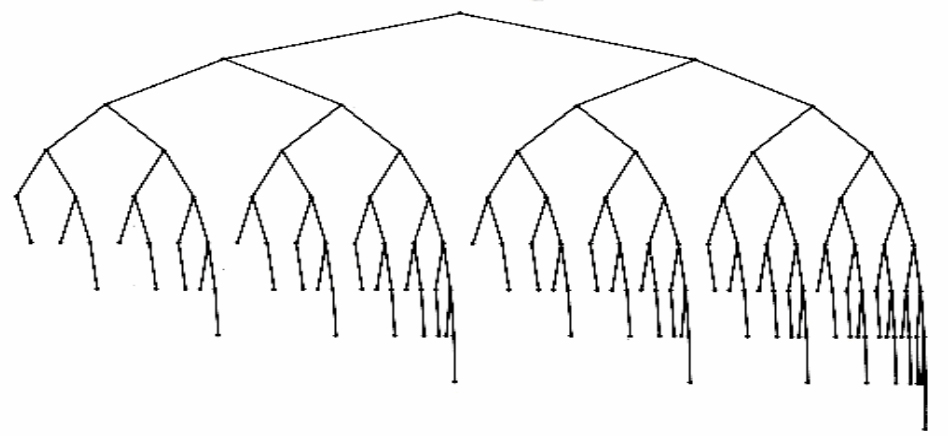


Figure 4.21 Smallest AVL tree of height 9

4.4.1 Operations of AVL trees

The basic operations of an AVL tree generally involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but modifications are preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

4.4.1.1 Insertion

Pictorial description of how rotations cause rebalancing tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

After inserting a node, it is necessary to check each of the node's ancestors for consistency with the rules of AVL. For each node checked, if the balance factor remains -1, 0, or 1 then no rotations are necessary. However, if the balance factor becomes 2 or -2 then the subtree rooted at this node is unbalanced. If insertions are performed serially, after each insertion, at most two tree rotations are needed to restore the entire tree to the rules of AVL.

There are four cases which need to be considered, of which two are symmetric to the other two. Let P be the root of the unbalanced subtree. Let R be the right child of P. Let L be the left child of P.

Right-Right case and **Right-Left case**: If the balance factor of P is -2, then the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must be checked. If the balance factor of R is -1, a **left rotation** is needed with P as the root. If the balance factor of R is +1, a **double left rotation** is needed. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

Left-Left case and **Left-Right case**: If the balance factor of P is +2, then the left subtree outweighs the right subtree of the given node, and the balance factor of the left child (L) must then checked. If the balance factor of L is +1, a **right rotation** is needed with P as the root. If the balance factor of L is -1, a **double right rotation** is needed. The first rotation is a left rotation with L as the root. The second is a right rotation with P as the root.

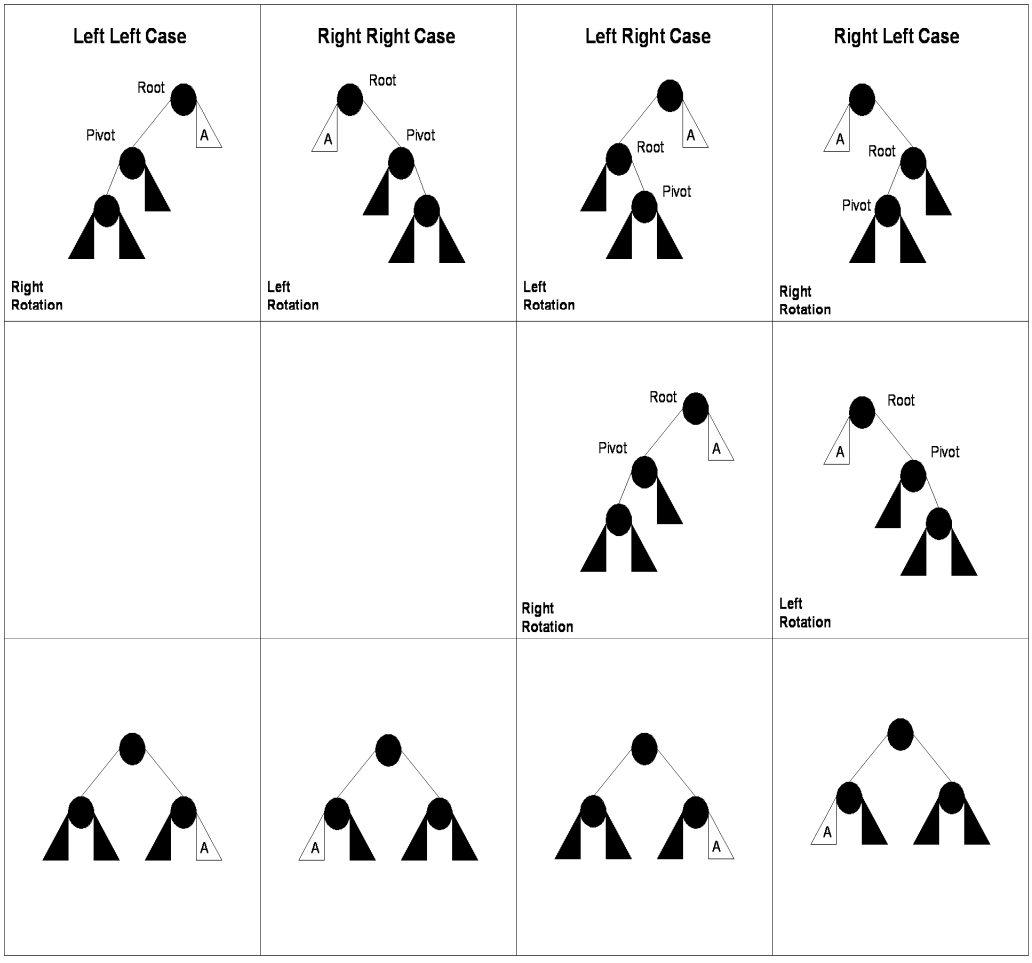


Figure 4.22 Insertion

4.4.1.2 Rotation Algorithms for putting an out-of-balance AVL-tree back in balance.

Note: After the insertion of each node the tree should be checked for balance. The only nodes that need checked are the ones along the insertion path of the newly inserted node. Once the tree is found to be out-of-balance then re-balance it using the appropriate algorithm. If the out-of-balance is detected as soon as it happens and the proper algorithm is used then the tree will be back in balance after one rotation.

Step 1: Set up the pointers:

A - points to the node that is out of balance. If more than one node is out of balance then select the one that is furthest from the root. If there is a tie then you missed a previous out-of-balance.

B – points to the child of A in the direction of the out-of-balance

C – points to the child of B in the direction of the out-=of-balance

F – points to the parent of A. This is the only pointer of these 4 that is allowed to be NULL.

Step 2: Determine the appropriate algorithm:

The first letter of the algorithm represents the direction from A to B (either **R**ight or **L**eft). The second letter represents the direction from B to C (either **R**ight or **L**eft).

Step 3: Follow the algorithm:

```
RR:                                 LL:
A.Right = B.Left                    A.Left = B.Right
B.Left = A                          B.Right = A
If F = NULL                         If F = NULL
        B is new Root of tree               B is new Root of tree
Else                                Else
        If F.Right = A                      If F.Right = A
            F.Right = B                             F.Right = B
        Else                                        Else
                F.Left = B                          F.Left = B
        End If                      End If
End If                              End If
```

```
RL:
B.Left = C.Right
A.Right = C.Left
C.Right = B
C.Left = A
If F = NULL
        C is new Root of tree
Else
        If F.Right = A
                F.Right = C
        Else
                F.Left = C
        End If
End If
```

```
LR:
A.Left = C.Right
B.Right = C.Left
C.Left = B
C.Right = A
If F = NULL
        C is new Root of tree
Else
        If F.Right = A
                F.Right = C
        Else
                F.Left = C
        End If
End If
```

Figure 4.23 Rotation Algorithms

4.4.1.3 Deletion

If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as replacement has at most one subtree. After deletion, retrace the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

In addition to the balancing described above for insertions, if the balance factor for the tree is 2 and that of the left subtree is 0, a right rotation must be performed on P. The mirror of this case is also necessary.

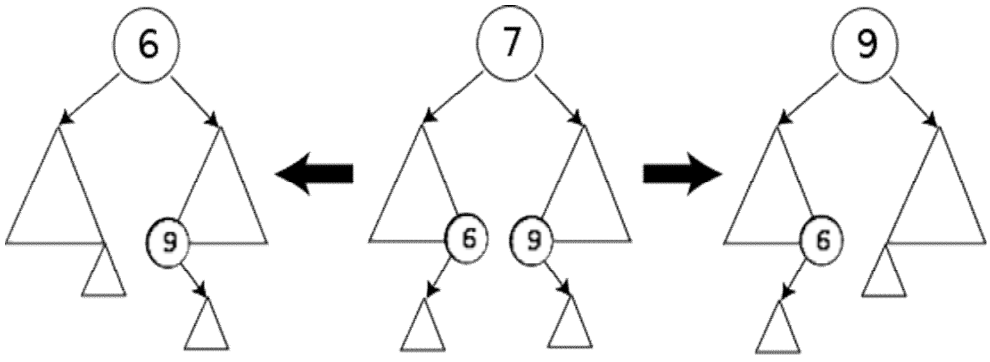


Figure 4.24 Deletion

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is O(log *n*) for lookup, plus a maximum of O(log *n*) rotations on the way back to the root, so the operation can be completed in O(log *n*) time.

4.4.1.4 Lookup

Lookup in an AVL tree is performed exactly as in an unbalanced binary search tree. Because of the height-balancing of the tree, a lookup takes O(log *n*) time. No special provisions need to be taken, and the tree's structure is not modified by lookups. (This is in contrast to splay tree lookups, which do modify their tree's structure.)

If each node additionally records the size of its subtree (including itself and its descendants), then the nodes can be retrieved by index in O(log *n*) time as well.

Once a node has been found in a balanced tree, the *next* or *previous* node can be obtained in amortized constant time. (In a few cases, about 2*log(n) links will need to be traversed. In most cases, only a single link needs to be traversed. On average, about two links need to be traversed.)

Check your progress:

1. Give a precise expression for the minimum number of nodes in an AVL tree of height *h*.
2. What is the minimum number of nodes in an AVL tree of height 15?
3. Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL tree.
4. Keys 1, 2, . . . , 2^k -1 are inserted in order into an initially empty AVL tree. Prove that the resulting tree is perfectly balanced.
5. Write the procedures to implement AVL single and double rotations.

6. Write a nonrecursive function to insert into an AVL tree.
7. How can you implement (nonlazy) deletion in AVL trees?
8. How many bits are required per node to store the height of a node in an n-node AVL tree?
9. What is the smallest AVL tree that overflows an 8-bit height counter?

4.5 SPLAY TREES:

A **splay tree** is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log(n)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Sleator and Robert Tarjan.

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

4.5.1 Advantages and disadvantages

Good performance for a splay tree depends on the fact that it is self-balancing, and indeed self optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. This is an advantage for nearly all practical applications, and is particularly useful for implementing caches and garbage collection algorithms.

Advantages include:

- Simple implementation -- simpler than other self-balancing binary search trees, such as red-black trees or AVL trees.
- Comparable performance -- average-case performance is as efficient as other trees.
- Small memory footprint -- splay trees do not need to store any bookkeeping data.

- Possible to create a persistent data structure version of splay trees -- which allows access to both the previous and new versions after an update. This can be useful in functional programming, and requires amortized O(log *n*) space per update.
- Work well with nodes containing identical keys -- contrary to other types of self balancing trees. Even with identical keys, performance remains amortized O(log *n*). All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the left most or right most node of a given key.

Disadvantages

- There could exist trees which perform "slightly" faster (a log(log(N))[?] factor) for a given distribution of input queries.
- Poor performance on uniform access (with workaround) -- a splay tree's performance will be considerably (although not asymptotically) worse than a somewhat balanced simple binary search tree for uniform access.

One worst case issue with the basic splay tree algorithm is that of sequentially accessing all the elements of the tree in the sorted order. This leaves the tree completely unbalanced (this takes n accesses, each an O(log *n*) operation). Reaccessing the first item triggers an operation that takes O(n) operations to rebalance the tree before returning the first item. This is a significant delay for that final operation, although the amortized performance over the entire sequence is actually O(log *n*). However, recent research shows that randomly rebalancing the tree can avoid this unbalancing effect and give similar performance to the other self-balancing algorithms.[

4.5.2 Operations

4.5.2.1 Splaying

When a node *x* is accessed, a splay operation is performed on *x* to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves *x* closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

Each particular step depends on three factors:

- Whether *x* is the left or right child of its parent node, *p*,
- Whether *p* is the root or not, and if not
- Whether *p* is the left or right child of *its* parent, *g* (the *grandparent* of x).

The three types of splay steps are:

Zig Step: This step is done when *p* is the root. The tree is rotated on the edge between *x* and *p*. Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when *x* has odd depth at the beginning of the operation.

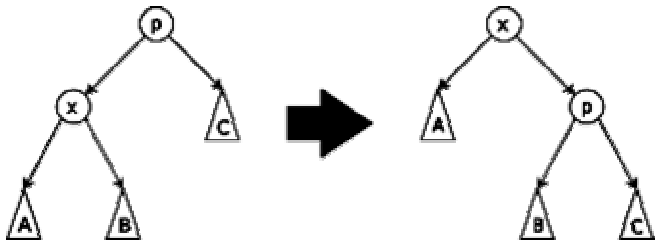


Figure 4.25 Zig Step

Zig-zig Step: This step is done when *p* is not the root and *x* and *p* are either both right children or are both left children. The picture below shows the case where *x* and *p* are both left children. The tree is rotated on the edge joining *p* with *its* parent *g*, then rotated on the edge joining *x* with *p*. Note that zig-zig steps are the only things that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro prior to the introduction of splay trees.

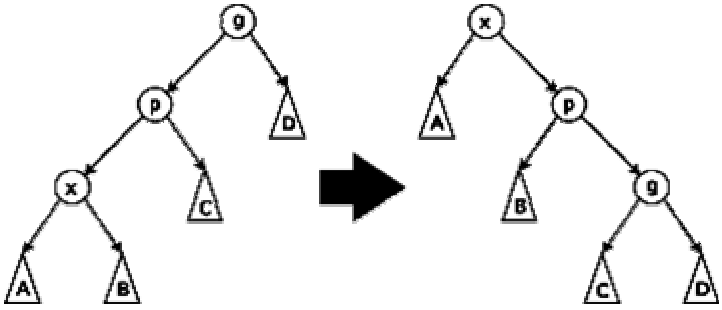


Figure 4.26 Zig-zig Step

Zig-zag Step: This step is done when *p* is not the root and *x* is a right child and *p* is a left child or vice versa. The tree is rotated on the edge between *x* and *p*, then rotated on the edge between *x* and its new parent *g*.

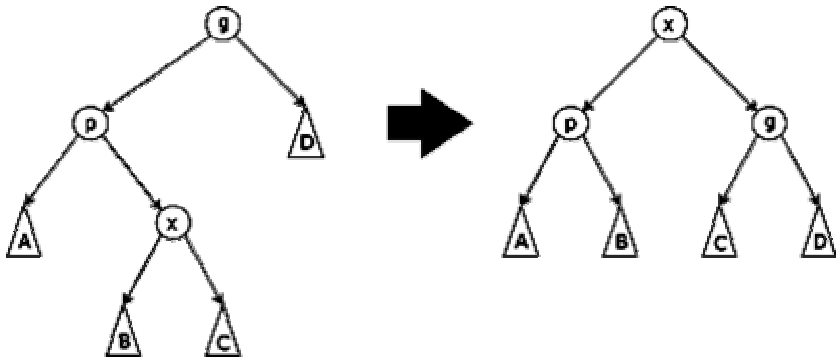


Figure 4.27 Zig-Zag Step

4.5.2.2 Insertion

The process of inserting a node x into a splay tree is *different* from inserting a node into a binary tree. The reason is that after insertion we want x to be the new root of the splay tree.

First, we search x in the splay tree. If x does not already exist, then we will not find it, but its parent node y. Second, we perform a splay operation on y which will move y to the root of the splay tree. Third, we insert the new node x as root in an appropriate way. In this way either y is left or right child of the new root x.

4.5.2.3 Deletion

The deletion operation for splay trees is somewhat different than for binary or AVL trees. To delete an arbitrary node *x* from a splay tree, the following steps can be followed:

1. Splay node *x*, sending it to the root of the tree.

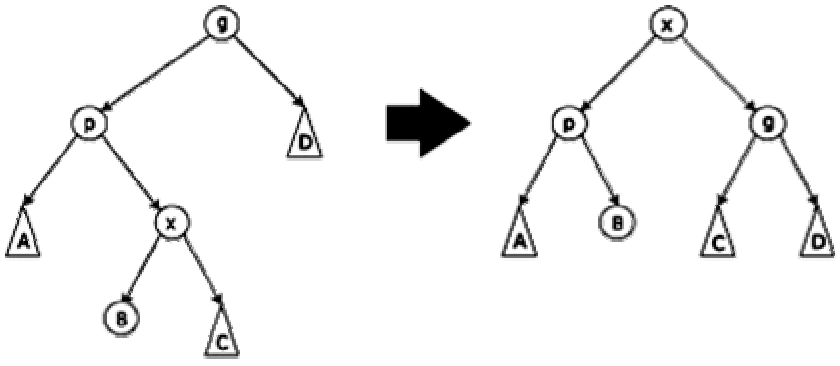


Figure 4.28 Deletion Step 1

2. Perform a left tree rotation on the first left child of *x* until the first left child of *x* has no right children.

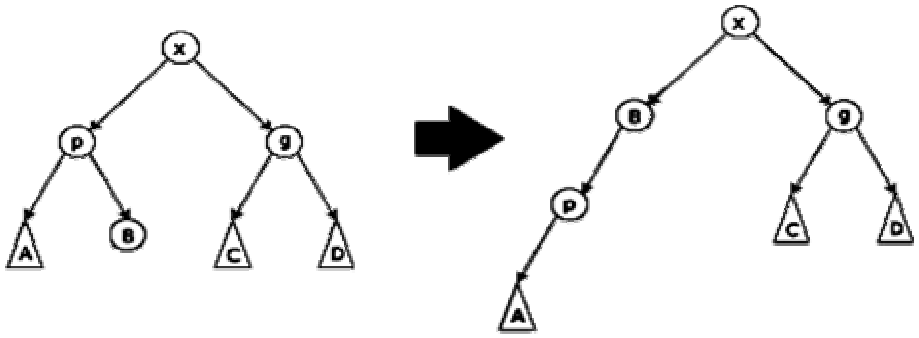


Figure 4.29 Deletion Step 2

3. Delete *x* from the tree & replace the root with the first left child of *x*.

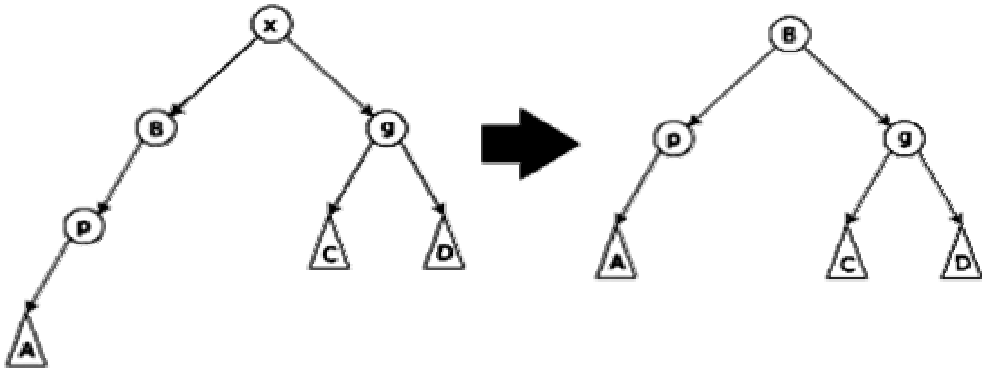
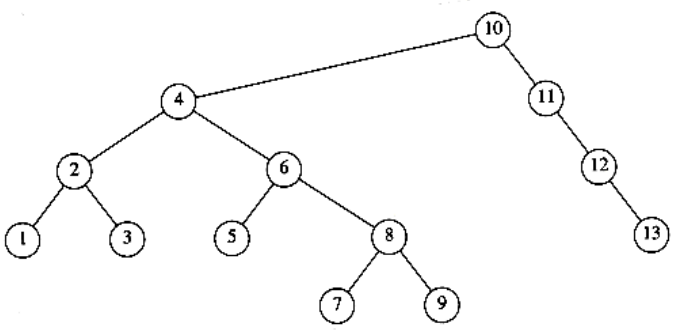


Figure 4.30 Deletion Step 3

Check your progress:

1. Show the result of accessing the keys 3, 9, 1, 5 in order in the splay tree in following Figure.



2. Nodes 1 through n = 1024 form a splay tree of left children.
    a. What is the internal path length of the tree (exactly)?
    b. Calculate the internal path length after each of find(1), find(2), find(3), find(4), find(5), find(6).
    c. If the sequence of successive finds is continued, when is the internal path length minimized?
3. Show that if all nodes in a splay tree are accessed in sequential order, the resulting tree consists of a chain of left children.
4. Write a program to perform random operations on splay trees. Count the total number of rotations performed over the sequence. How does the running time compare to AVL trees and unbalanced binary search trees?

4.6 THE SEARCH TREE ADT-BINARY SEARCH TREES

An important application of binary trees is their use in searching. Let us assume that each node in the tree is assigned a key value. In our examples, we will assume for simplicity that these are integers, although arbitrarily complex keys are allowed. We will also assume that all the keys are distinct.

The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X, and the values of all the keys in the right subtree are larger than the key value in X. Notice that this implies that all the elements in the tree can be ordered in some consistent manner. In Figure 4.31, the tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with key 7 in the left subtree of a node with key 6 (which happens to be the root).

Figure 4.31 Two binary trees (only the left tree is a search tree)

We now give brief descriptions of the operations that are usually performed on binary search trees. Note that because of the recursive definition of trees, it is common to write these routines

recursively. Because the average depth of a binary search tree is $O(\log n)$, we generally do not need to worry about running out of stack space. We repeat our type definition in Figure 4.32. Since all the elements can be ordered, we will assume that the operators <, >, and = can be applied to them, even if this might be syntactically erroneous for some types.

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
        element_type element;
        tree_ptr left;
        tree_ptr right;
};
typedef tree_ptr SEARCH_TREE;
```

Figure 4.32 Binary search tree declarations

4.6.1. Make_null

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely. It is also a simple routine, as evidenced by Figure 4.33

```
SEARCH_TREE
make_null ( void )
{
        return NULL;
}
```

Figure 4.33 Routine to make an empty tree

4.6.2. Find

This operation generally requires returning a pointer to the node in tree *T* that has key *x*, or *NULL* if there is no such node. The structure of the tree makes this simple. If *T* is *, then we can just return* . Otherwise, if the key stored at *T* is *x*, we can return *T*. Otherwise, we make a recursive call on a subtree of *T*, either left or right, depending on the relationship of *x* to the key stored in *T*. The code in Figure 4.34 is an implementation of this strategy.

```
tree_ptr
find( element_type x, SEARCH_TREE T )
{
        if( T == NULL )
                return NULL;
        if( x < T->element )
                return( find( x, T->left ) );
        else
                if( x > T->element )
                    return( find( x, T->right ) );
                else
                    return T;
}
```

Figure 4.34 Find operation for binary search trees

Notice the order of the tests. It is crucial that the test for an empty tree be performed first, since otherwise the indirections would be on a *NULL* pointer. The remaining tests are arranged with the least likely case last. Also note that both recursive calls are actually tail recursions and can be easily removed with an assignment and a *goto*. The use of tail recursion is justifiable here because the simplicity of algorithmic expression compensates for the decrease in speed, and the amount of stack space used is expected to be only $O(\log n)$.

4.6.3. Find_min and find_max

These routines return the position of the smallest and largest elements in the tree, respectively. Although returning the exact values of these elements might seem more reasonable, this would be inconsistent with the *find* operation. It is important that similar-looking operations do similar things. To perform a *find_min*, start at the root and go left as long as there is a left child. The stopping point is the smallest element. The *find_max* routine is the same, except that branching is to the right child.

This is so easy that many programmers do not bother using recursion. We will code the routines both ways by doing *find_min* recursively and *find_max* nonrecursively (see Figs. 4.35 and 4.36).

Notice how we carefully handle the degenerate case of an empty tree. Although this is always important to do, it is especially

crucial in recursive programs. Also notice that it is safe to change *T* in *find_max*, since we are only working with a copy. Always be extremely careful, however, because a statement such as *T -> right* : *=T -> right -> right* will make changes in most languages.

```
tree_ptr
find_min( SEARCH_TREE T )
{
        if( T == NULL )
                return NULL;
        else
                if( T->left == NULL )
                        return( T );
                else
                        return( find_min ( T->left ) );
}
```

Figure 4.35 Recursive implementation of find_min for binary search trees

```
tree_ptr
find_max( SEARCH_TREE T )
{
        if( T != NULL )
                while( T->right != NULL )
                        T = T->right;
        return T;
}
```

Figure 4.36 Nonrecursive implementation of find_max for binary search trees

4.6.4. Insert

The insertion routine is conceptually simple. To insert *x* into tree *T*, proceed down the tree as you would with a *find*. If *x* is found, do nothing (or "update" something). Otherwise, insert *x* at the last spot on the path traversed. Figure 4.37 shows what happens. To insert 5, we traverse the tree as though a *find* were occurring. At the node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct spot.

Duplicates can be handled by keeping an extra field in the node record indicating the frequency of occurrence. This adds some extra space to the entire tree, but is better than putting duplicates in the tree (which tends to make the tree very deep). Of course this strategy does not work if the key is only part of a larger record. If that is the case, then we can keep all of the records that

have the same key in an auxiliary data structure, such as a list or another search tree.

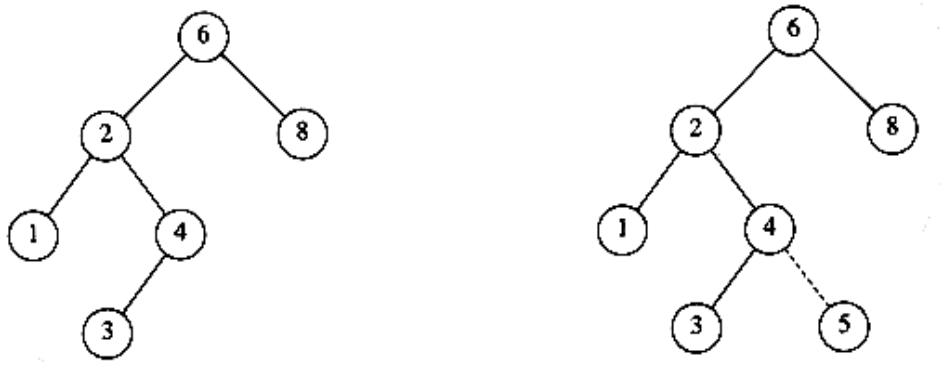


Figure 4.37 Binary search trees before and after inserting 5

Figure 4.38 shows the code for the insertion routine. Since *T* points to the root of the tree, and the root changes on the first insertion, *insert* is written as a function that returns a pointer to the root of the new tree. Lines 8 and 10 recursively insert and attach *x* into the appropriate subtree.

```
tree_ptr
insert( element_type x, SEARCH_TREE T )
{
/*1*/       if( T == NULL )
{  /* Create and return a one-node tree */
/*2*/           T = (SEARCH_TREE) malloc ( sizeof (struct tree_node)
);
/*3*/           if( T == NULL )
/*4*/               fatal_error("Out of space!!!");
else
{
/*5*/               T->element = x;
/*6*/               T->left = T->right = NULL;
}
}
else
/*7*/       if( x < T->element )
/*8*/           T->left = insert( x, T->left );
else
/*9*/       if( x > T->element )
/*10*/           T->right = insert( x, T->right );
/* else x is in the tree already. We'll do nothing */
/*11*/      return T; /* Don't forget this line!! */
}
```

Figure 4.38 Insertion into a binary search tree

4.6.5. Delete

As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity). See Figure 4.39. Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved.

The complicated case deals with a node with two children. The general strategy is to replace the key of this node with the smallest key of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right subtree cannot have a left child, the second *delete* is an easy one. Figure 4.40 shows an initial tree and the result of a deletion. The node to be deleted is the left child of the root; the key value is 2. It is replaced with the smallest key in its right subtree (3), and then that node is deleted as before.

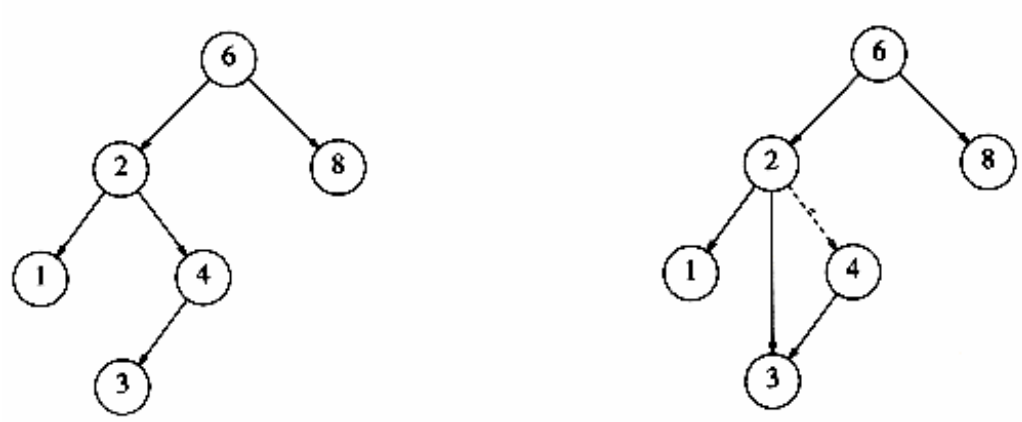


Figure 4.39 Deletion of a node (4) with one child, before and after

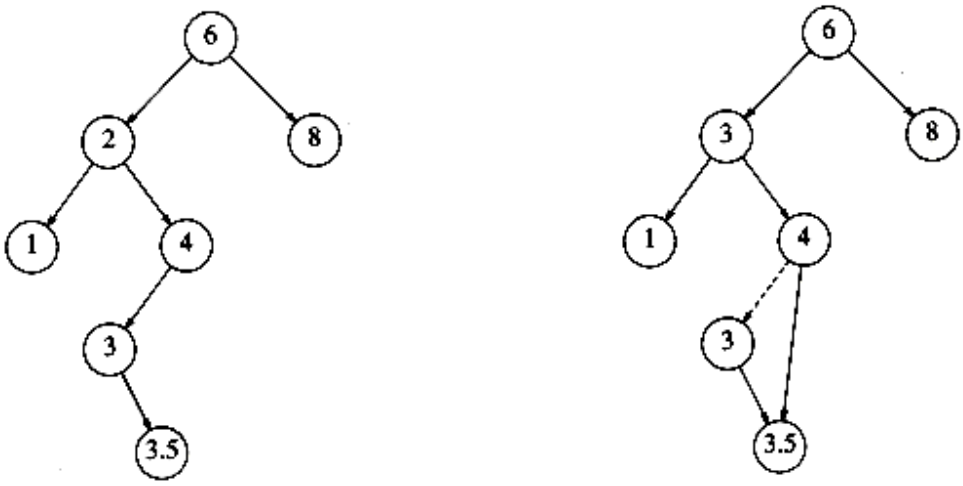


Figure 4.40 Deletion of a node (2) with two children, before and after

The code in Figure 4.41 performs deletion. It is inefficient, because it makes two passes down the tree to find and delete the smallest node in the right subtree when this is appropriate. It is easy to remove this inefficiency, by writing a special *delete_min* function, and we have left it in only for simplicity.

If the number of deletions is expected to be small, then a popular strategy to use is *lazy deletion:* When an element is to be deleted, it is left in the tree and merely *marked* as being deleted. This is especially popular if duplicate keys are present, because then the field that keeps count of the frequency of appearance can be decremented. If the number of real nodes in the tree is the same as the number of "deleted" nodes, then the depth of the tree is only expected to go up by a small constant, so there is a very small time penalty associated with lazy deletion. Also, if a deleted key is reinserted, the overhead of allocating a new cell is avoided.

```
tree_ptr
delete( element_type x, SEARCH_TREE T )
{
        tree_ptr tmp_cell, child;
        if( T == NULL )
                error("Element not found");
        else
                if( x < T->element )  /* Go left */
                        T->left = delete( x, T->left );
                else
```

```
                    if( x > T->element )  /* Go right */
                        T->right = delete( x, T->right );
                    else      /* Found element to be deleted */
                    if( T->left && T->right )  /* Two children */
                    {
                    /* Replace with smallest in right subtree */
                        tmp_cell = find_min( T->right );
                        T->element = tmp_cell->element;
                        T->right = delete( T->element, T->right );
                    }
                    else      /* One child */
                    {
                        tmp_cell = T;
                        if( T->left == NULL )
                        /* Only a right child */
                        child = T->right;
                        if( T->right == NULL )
                        /* Only a left child */
                        child = T->left;
                        free( tmp_cell );
                        return child;
                    }
        return T;
}
```

Figure 4.41 Deletion routine for binary search trees

Check your progress:

1. Show the result of inserting 3, 1, 4, 6, 9, 2, 5, 7 into an initially empty binary search tree.
2. Show the result of deleting the root.
3. Write routines to implement the basic binary search tree operations.
4. Binary search trees can be implemented with cursors, using a strategy similar to a cursor linked list implementation. Write the basic binary search tree routines using a cursor implementation.

5. Write a program to evaluate empirically the following strategies for deleting nodes with two children:

   a. Replace with the largest node, X, in T_L and recursively delete X.

   b. Alternately replace with the largest node in T_L and the smallest node in T_R, and recursively delete appropriate node.

   c. Replace with either the largest node in T_L or the smallest node in T_R (recursively deleting the appropriate node), making the choice randomly. Which strategy seems to give the most balance? Which takes the least CPU time to process the entire sequence?

6. Write a routine to list out the nodes of a binary tree in *level-order*. List the root, then nodes at depth 1, followed by nodes at depth 2, and so on.

Let us Sum up:

We have seen uses of trees in operating systems, compiler design, and searching. Expression trees are a small example of a more general structure known as a *parse tree*, which is a central data structure in compiler design. Parse trees are not binary, but are relatively simple extensions of expression trees (although the algorithms to build them are not quite so simple).

Search trees are of great importance in algorithm design. They support almost all the useful operations, and the logarithmic average cost is very small. Nonrecursive implementations of search trees are somewhat faster, but the recursive versions are sleeker, more elegant, and easier to understand and debug. The problem with search trees is that their performance depends heavily on the input being random. If this is not the case, the running time increases significantly, to the point where search trees become expensive linked lists.

We saw several ways to deal with this problem. AVL trees work by insisting that all nodes' left and right subtrees differ in heights by at most one. This ensures that the tree cannot get too deep. The operations that do not change the tree, as insertion does, can all use the standard binary search tree code. Operations that change the tree must restore the tree. This can be somewhat complicated, especially in the case of deletion.

We also examined the splay tree. Nodes in splay trees can get arbitrarily deep, but after every access the tree is adjusted in a somewhat mysterious manner.

In practice, the running time of all the balanced tree schemes is worse (by a constant factor) than the simple binary search tree, but this is generally acceptable in view of the protection being given against easily obtained worst-case input.

A final note: By inserting elements into a search tree and then performing an inorder traversal, we obtain the elements in sorted order. This gives an $O(n \log n)$ algorithm to sort, which is a worst-case bound if any sophisticated search tree is used.

References:

1. G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Doklady* 3.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
3. B. Allen and J. I. Munro, "Self Organizing Search Trees," *Journal of the ACM*, 25 (1978).
4. R. A. Baeza-Yates, "Expected Behaviour of B^+- trees under Random Insertions," *Acta Informatica* 26 (1989).
5. R. A. Baeza-Yates, "A Trivial Algorithm Whose Analysis Isn't: A Continuation," *BIT* 29 (1989).
6. R. Bayer and E. M. McGreight, "Organization and Maintenance of Large Ordered Indices," *Acta Informatica* 1 (1972).
7. J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM* 18 (1975).
8. Data structure – A Pseudocode Approach with C – Richard F Gilberg Behrouz A. Forouzan, Thomson
9. Schaum's Outlines Data structure Seymour Lipschutz Tata McGraw Hill 2nd Edition
10. Data structures & Program Design in C Robert Kruse, C. L.Tondo, Bruce Leung Pearson
11. "Data structure using C" AM Tanenbaum, Y Langsam & M J Augustein, Prentice Hall India

Question Pattern

1. What are trees? Explain the different methods of tree traversals.
2. What are expression trees? Explain with examples.
3. Explain the AVL tree in detail.
4. What are splay trees? Explain in detail.
5. Explain the Binary search trees in detail.

5

HEAP

Unit Structure:

5.1 Introduction
5.2 Definition of a Heap
5.3 Types of Heap –
5.4 Representation of Heap in memory
5.5 Heap Algorithms

5.1. INTRODUCTION:

i. A heap is an efficient semi-ordered data structure for storing a collection of orderable data.

ii. The main difference between a heap and a binary tree is the heap property. In order for a data structure to be considered a heap, it must satisfy the heap property which is discussed in the further section

5.2. DEFINITION :

A Heap data structure is a binary tree with the following properties:

i. It is a ***complete binary tree***; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

ii. It satisfies the ***heap-order property***: The data item stored in each node is greater than or equal to the data items stored in its children.

ie. If *A* and *B* are elements in the heap and *B* is a child of *A*, then key(*A*) ≥ key(*B*).

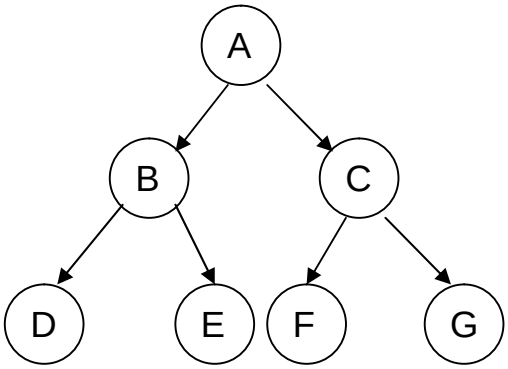


Fig 5.1

The above Tree is a Complete Binary tree and is a HEAP if we assume that the data item stored in each node is greater than or equal to the data items stored in its children

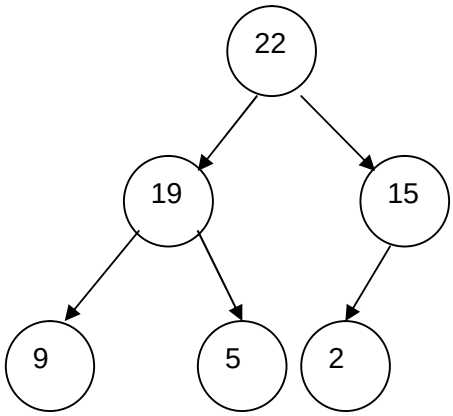


Fig 5.2

The above tree is a heap because

1) it is a complete binary tree
2) it satisfies the heap order property of heap

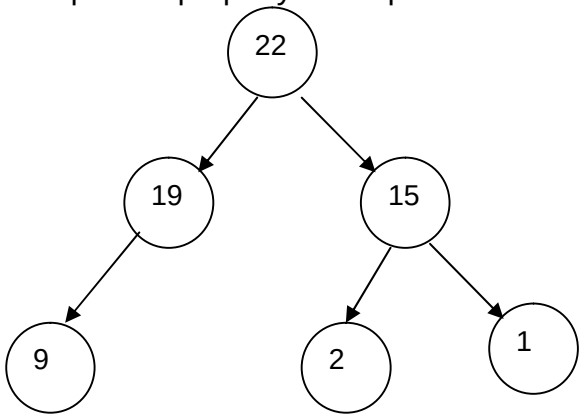


Fig 5.3

The above tree is not a heap since it is not a complete binary tree.

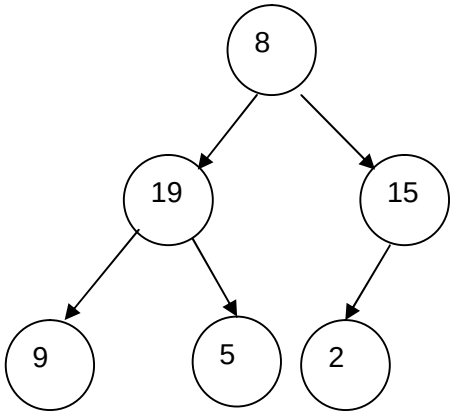


Fig 5.4

The above tree is not a heap since it does not satisfy the heap order property.i.e the value of data the root is less than the value at its child nodes.

5.3. TYPES OF HEAPS:

Depending upon the value of data items in the nodes of the binary tree, a heap could be of the following two types:

1) max-heap

2) min-heap

5.3.1 Max-Heap: A *max-heap* is often called as a Heap.

Definition
A max-heap is a binary tree structure with the following properties:

1) The tree is complete or nearly complete.
2) The key value of each node is greater than or equal to the key value of its children.
   .i.e If *A* and *B* are elements in the heap and *B* is a child of *A*, then key(*A*) ≥ key(*B*).

5.3.2 Min-Heap

Definition
A min-heap is a binary tree structure with the following properties:

1) The tree is complete or nearly complete.
2) The key value of each node is less than or equal to the key value of its children.

.i.e If *A* and *B* are elements in the heap and *B* is a child of *A*, then key(*A*) ≤ key(*B*).

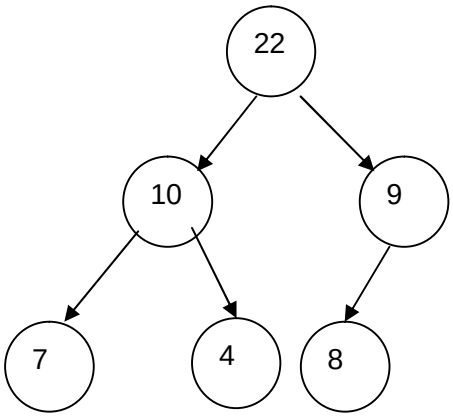


Fig 5.5

Max-heap (or heap)

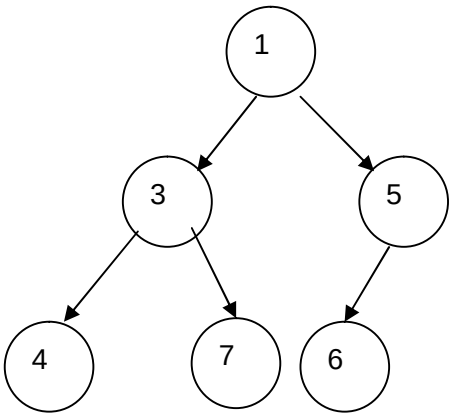


Fig 5.6

min-heap

5.4. REPRESENTATION OF HEAP IN MEMORY:

Heaps are represented in memory sequentially by means of an array.

Consider the following diagrams :

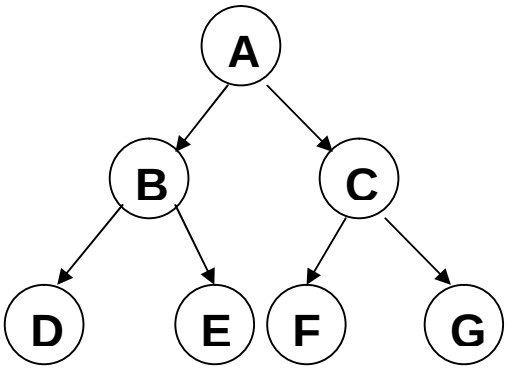


Fig 5.7.a

CONCEPTUAL Representation

1	2	3	4	5	6	7
A	B	C	D	E	F	G

Fig 5.7.b

PHYSICAL representation (in memory)

The root of the tree is TREE [1].

The left and right child of any node TREE [K] are TREE[2K] & TREE[2K+1] respectively

As in the diagram above,

- TREE [1] is the root node which contains A. (K=1)
- Left Node of B i.e left node of TREE [2] is given by TREE [2K] which is TREE [2] = B
- Right Node of A i.e. right node of TREE [1] is given by TREE [2K+1] which is TREE [3] = C

This representation could also be shown as

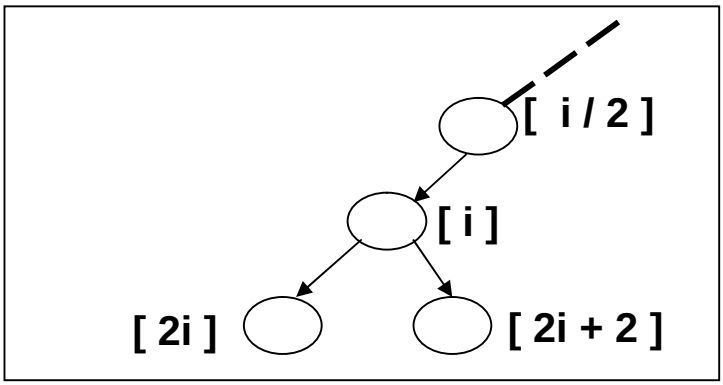


Fig 5.8

Note : In case the root node is Tree[0] i.e K=0, left node is given by 2K+1 and Right node is given by 2K+2

5.5. BASIC HEAP ALGORITHMS

5.5.1 ReheapUp: repairs a "broken" heap by floating the last element up the tree until it is in its correct location.

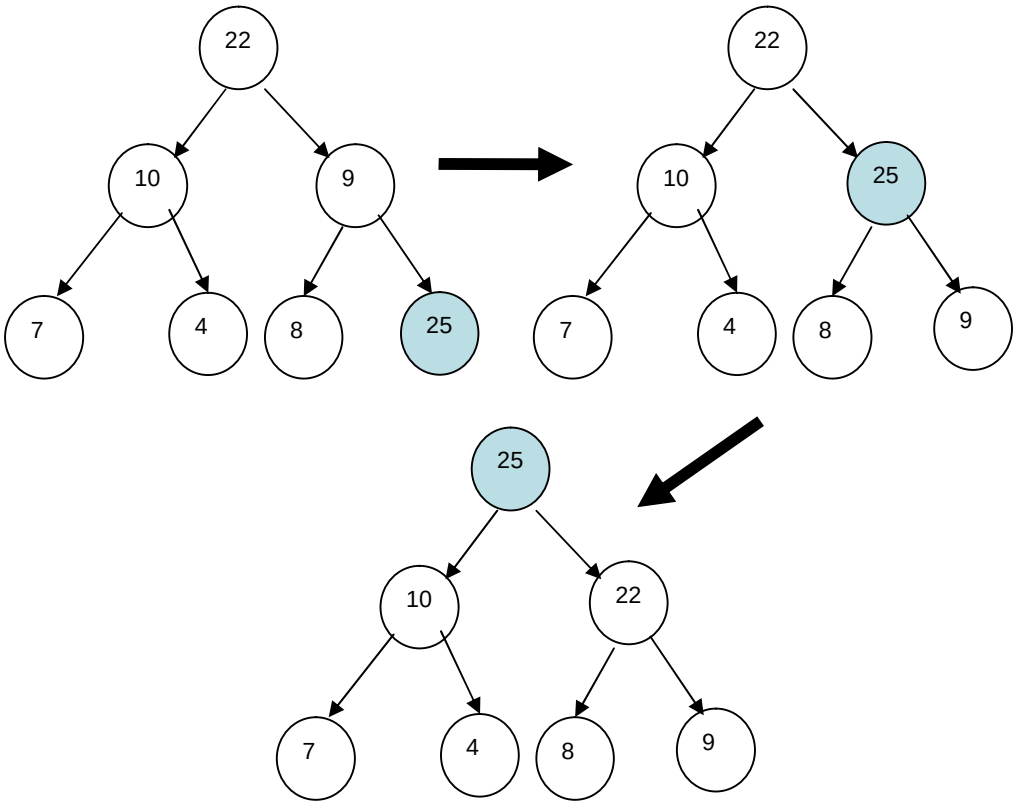


Fig 5.9 ReHeapUp

Algorithm ReheapUp(val position<int>)

This algorithm re-establishes heap by moving data in position up to its correct location. It is assumed that all data in the heap above this position satisfy key value order of a heap, except the data in position.

After execution Data in position has been moved up to its correct location.
The function ReheapUp is recursively used.

1. if (position<> 0)// *the parent of position exists.*
    1. parent = (position-1)/2
    2. if (data[position].key > data[parent].key)
        1. swap(position, parent) // *swap data at position with data at parent.*
        2. ReheapUp(parent)
2. return

EndReheapUp

5.5.2 Reheapdown

Repairs a "broken" heap by pushing the root of the sub-tree down until it is in its correct location.

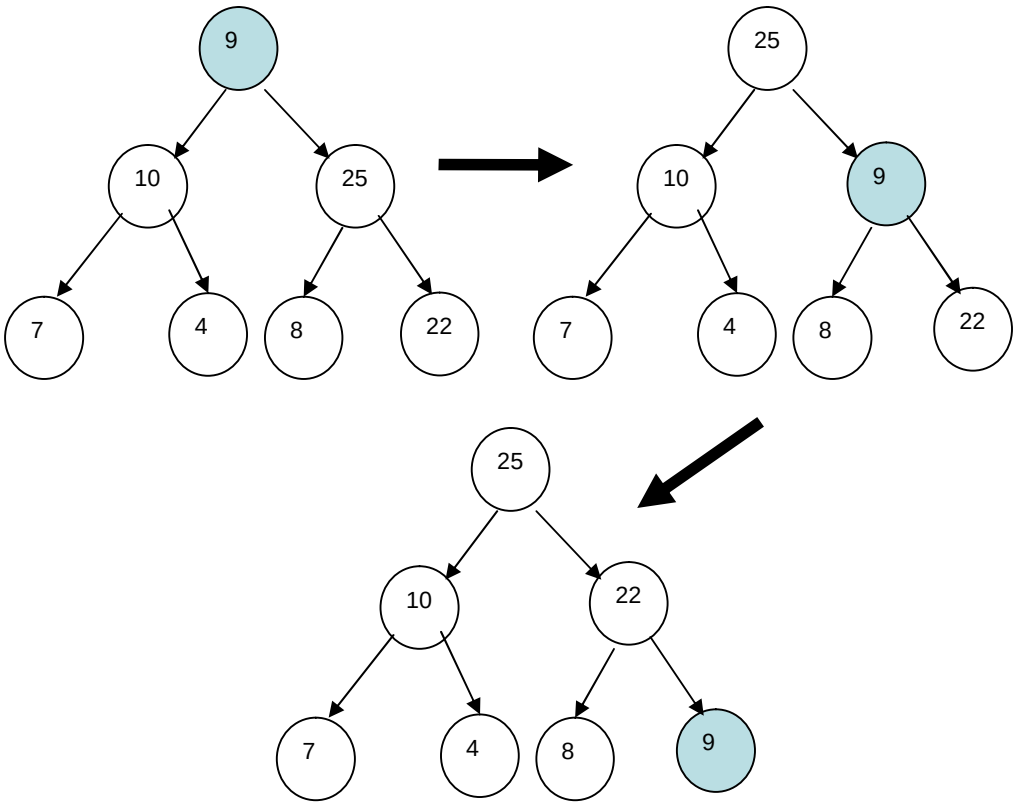


Fig 5.10 ReHeapDown

Algorithm: ReheapDown(val position<int>, val lastPosition<int>)

This algorithm re-establishes heap by moving data in position down to its correct location. It is assumed that all data in the sub-tree of position satisfy key value order of a heap, except the data in position.

After execution Data in position has been moved down to its correct location.

The function ReheapDown is recursively used.

1. leftChild= position*2 + 1

2. rightChild= position*2 + 2

3. if (leftChild<= lastPosition) *// the left child of position exists.*

1. if (rightChild<= lastPosition) AND (data[rightChild].key > data[leftChild].key)

    1. child = rightChild

2. else

    1. child = leftChild // *choose larger child to compare with data in position*

3. if (data[child].key > data[position].key)

    1. swap(child, position) // *swap data at position with data at child.*

    2. ReheapDown(child, lastPosition)

4. return

End ReheapDown

5.5.3 Inserting into a Heap

- A new node (say key k) is always inserted at the end of the heap. After the insertion of a new key k, the heap-order property may be violated.
- After the insertion of the new key k, the heap-order property may be violated.
- Algorithm heapUp restores the heap-order property by swapping k along an upward path from the insertion node
- heapUp terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k.

Algorithm: InsertHeap(val DataIn<DataType>) // *Recursive version.*

This algorithm inserts new data into the min-heap.

After execution DataIn has been inserted into the heap and the heap order property is maintained.

The algorithm makes use of the recursive function ReheapUp.

1. if(heap is full)

    1. return *overflow*

2. else

    1. data[count] = DataIn

    2. ReheapUp(count)

3. count= count+ 1

4. return *success*

End InsertHeap

5.5.4 Deleting from a Heap

- Removes the minimum element from the min-heap .i.e. root
- The element in the last position (say k) replaces the root.
- After the deletion of the root and its replacement by k, the heap-order property may be violated.
- Algorithm heapDown restores the heap-order property by swapping key k along a downward path from the root
- heapDown terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k

DeleteHeap(ref MinData<DataType>)
This algorithm removes the minimum element from the min-heap.

After execution MinData receives the minimum data in the heap and this data has been removed and the heap has been rearranged.

The algorithm makes use of the recursive function ReheapDown.

1. if(heap is empty)

    1. return *underflow*

2. else

    1. MinData= Data[0]

    2. Data[0] = Data[count -1]

    3. count= count-1

    4. ReheapDown(0, count -1)

    5. return *success*

End DeleteHeap

5.5.5 Building a Heap

BuildHeap(val listOfData<List>)

The following algorithm builds a heap from data from listOfData.

It is assumed that listOfData contains data that need to be inserted into an empty heap.

The algorithm makes use of the recursive function ReheapUp.

1. count= 0
2. repeat while (heap is not full) AND (more data in listOfData)
    1. listOfData.Retrieve(count, newData)
    2. data[count] = newData
    3. ReheapUp(count)
    4. count= count+ 1
3. if (count< listOfData.Size())
    1. return overflow
4. else
    1. return success

End BuildHeap

Example:

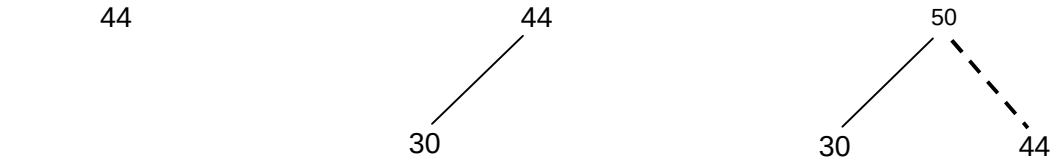
Build a heap from the following list of numbers

44, 30, 50, 22, 60, 55, 77, 55

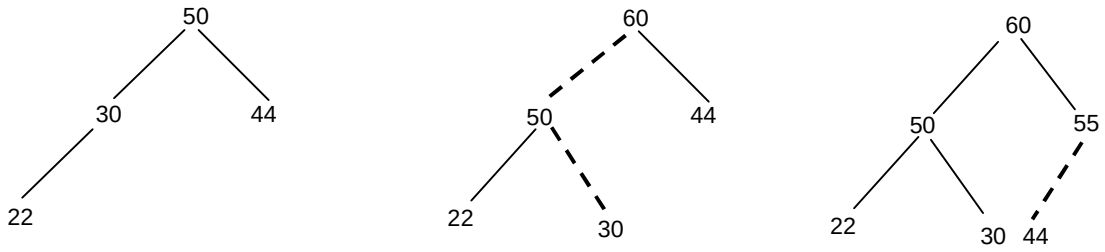
This can be done by inserting the eight elements one after the other into an empty heap H.

Fig 5.11.a to 6.11.h shows the respective pictures of the heap after each of the eight elements has been inserted.

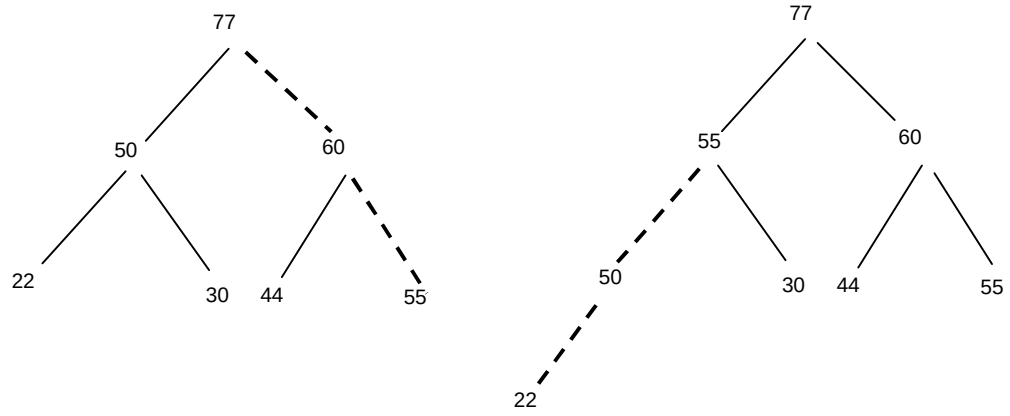
The dotted lines indicate that an exchange has taken place during the insertion of the given item of information.



a. ITEM = 44 b. ITEM = 30 c. ITEM = 50



d. ITEM = 22 e. ITEM = 60 f. ITEM = 55



g. ITEM = 77 h. ITEM = 55

Fig 5.11 Building a heap

6

MULTIWAY TREES

Unit Structure:

6.1 Definition of a Multi-way Tree
6.2 B-Trees

6.1. DEFINITION OF MULTI-WAY TREE:

A Multi-way tree is a tree which each node contains one or more keys.

A Multi-way (or m-way) search tree of order m is a tree in which

- each node has m or less than m sub-trees and
- contains one less key than its sub-trees
- e.g. The figure below is a multi-way search tree of order 4 because
  - the root has 4 sub-trees and
  - It contains 3 keys (.i.e. root contains one less key than no. of sub-trees)

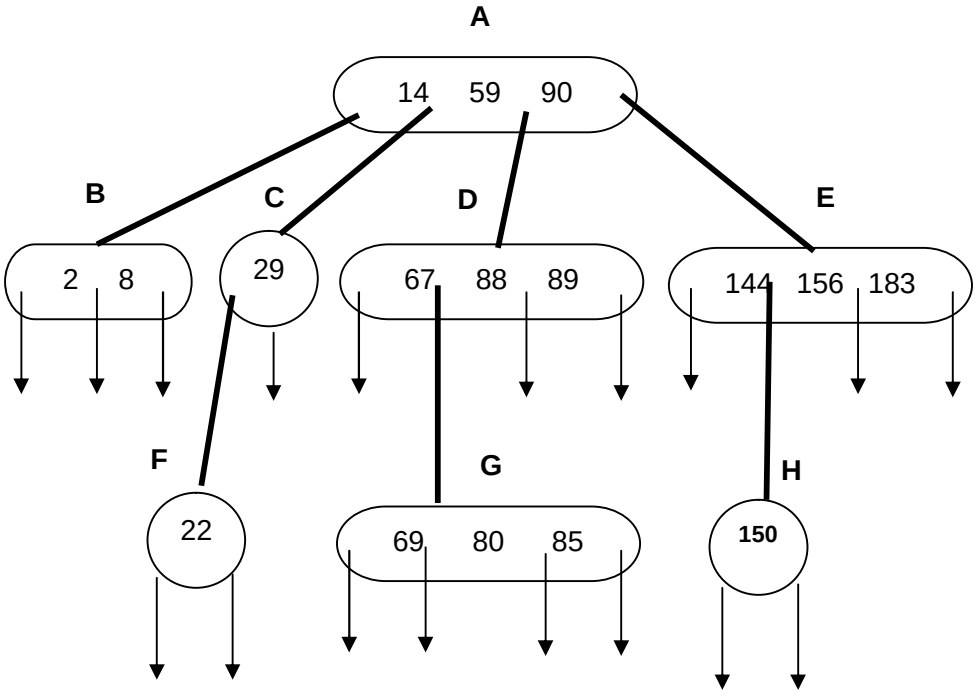


Fig.6.1. Multi-way search tree

- The keys in each node are sorted
- All the keys in a sub-tree S_0 are less than or equal to K_0
  - .e.g. keys of the sub-tree B (.i.e. 2 and 8) are less than first key of the root A (.i.e.14)
- All keys in the sub-tree Sj (1 < j < m-2) are greater than Kj-1 and less than or equal to Kj
  - .e.g. the second sub-tree of A contains the keys 29 & 22 which are greater than 14 (the first key of A) and less than 59 (the second key of A)

6.2 B-TREES :

6.2.1 Introduction

A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fair amount of data at once.

6.2.2 Definition :

A B-tree of order m is a multi-way search tree of order m such that:

- All leaves are on the bottom level.
- Each non-root node contains at least (m-1)/2 keys
- Each node can have at most m children
- For each node, if k is the actual number of children in the node, then k - 1 is the number of keys in the node

A B-Tree that is a multi-way search tree of order 4 has to fulfill the following conditions related to the ordering of the keys:

- The keys in each node are in ascending order.
- At the root node the following is true:
  - The sub-tree starting at first branch of the root has only keys that are less than the first key of the root

- The sub-tree starting at the second branch of root has only keys that are greater than first key of root and at the same time less than second key of root.
- The sub-tree starting at third branch has only keys that are greater than second key of root and at the same time less than third key of root.
- The sub-tree starting at the last branch .i.e. the fourth branch has only keys that are greater than the third key of root.

Example

The following is an example of a B-tree of order 4.

This means that all non root nodes must have at least (4 -1)/ 2 = 1 children

Of course, the maximum number of children that a node can have is 4 (so 3 is the maximum number of keys).

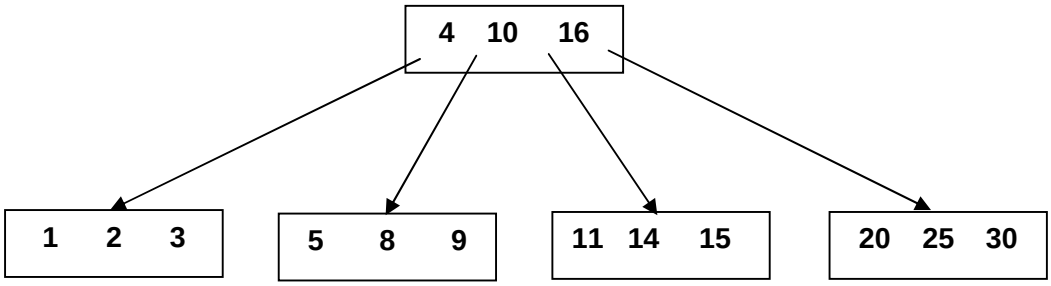


Fig.6.2: B Tree of order 4

In the above diagram the following are true:

- The sub-tree starting at first branch of the root has the keys 1,2,3 that are less than the first key of the root .i.e. 4
- The sub-tree starting at the second branch of root has the keys 5, 8, 9 that are greater than first key of root which is 4 and at the same time less than second key of root which is 10.
- The sub-tree starting at third branch has the keys 11, 14, 15 that are greater than second key of root which is 10 and at the same time less than third key of root which is 16.
- The sub-tree starting at the last branch .i.e. the fourth branch has the keys 20, 25, 30 that are greater than the third key of root which is 16.

6.2.3 B- Tree Operations:

In this section we will study the following operations

1. Insertion in a B-Tree
2. Deletion in a B-Tree

6.2.3.1 Insertion in a B-Tree:

In a B-Tree Insertion takes place at the leaf node

1. Search for the leaf node where the data is to be entered

2. If the node has less than m-1 entries then the new data are simply inserted in the sequence.

3. If the node is full, the insertion causes overflow. Therefore split the node into two nodes. Bring the median (middle) data to its parents left entries of the median to be copied to the left sub-tree and right entries copied into right sub-tree.

While splitting we come across the following circumstances

- The new key is less than the median key
- The new key is the median key
- The new key is greater than the median key

If the new key is less than or equal to the median key the new data belongs to the left or original node.

If the new key is greater than the median key, the data belongs to the new node

Example:

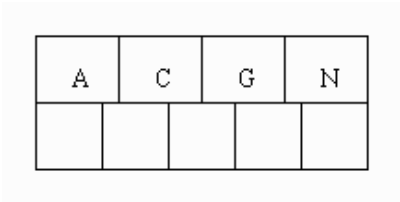
Construct a B-Tree of order 5 which is originally empty from the following data:

C N G A H E K Q M F W L T Z D P R X Y S

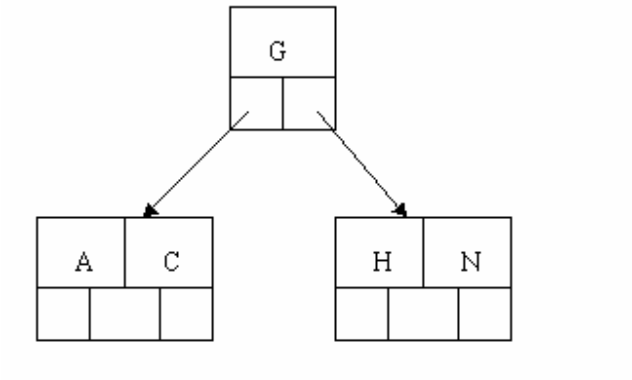
Order 5 means that a node can have a maximum of 5 children and 4 keys.

All nodes other than the root must have a minimum of 2 keys.

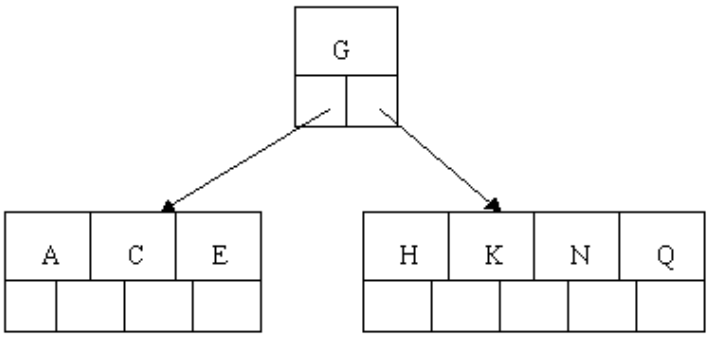
The first 4 letters get inserted into the same node, resulting in this picture



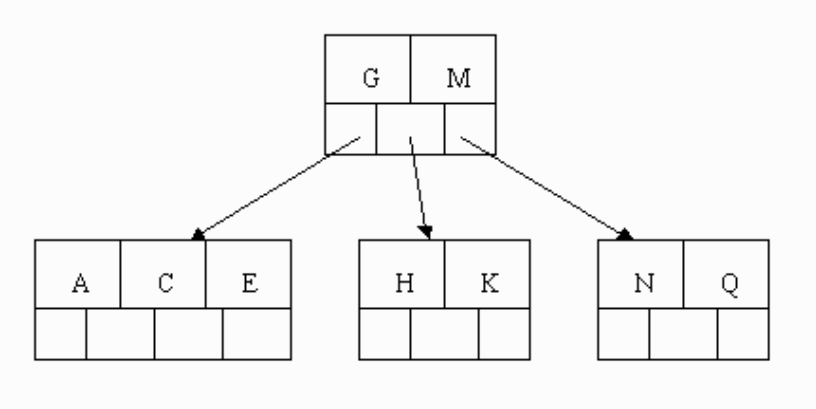
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



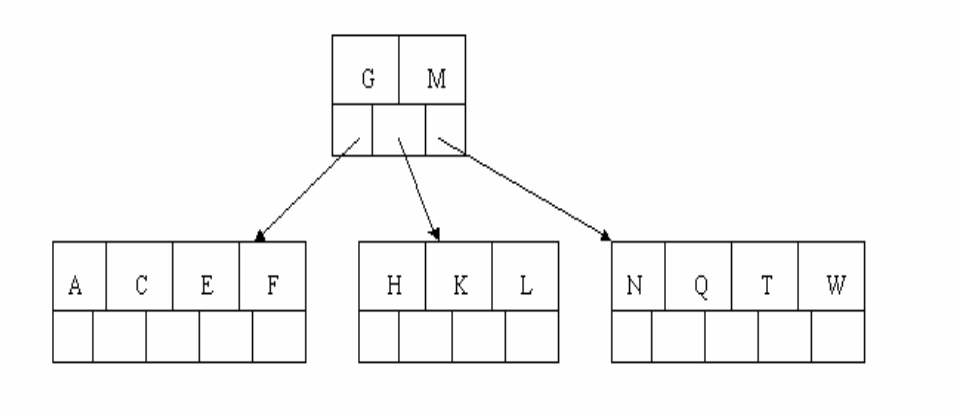
Inserting E, K, and Q proceeds without requiring any splits:



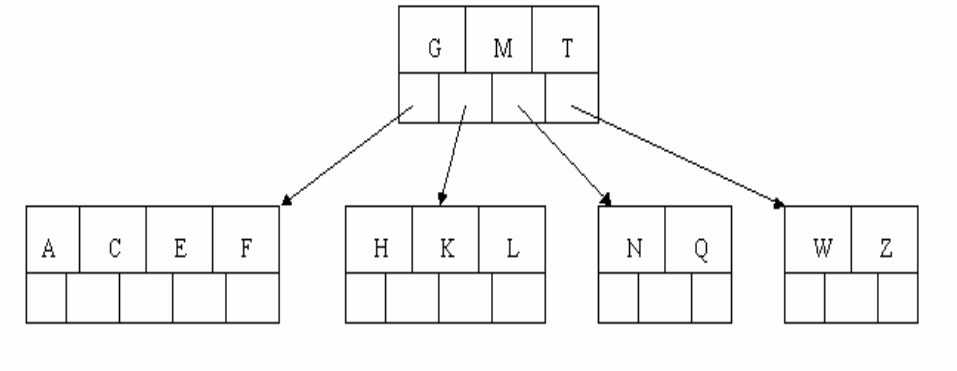
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



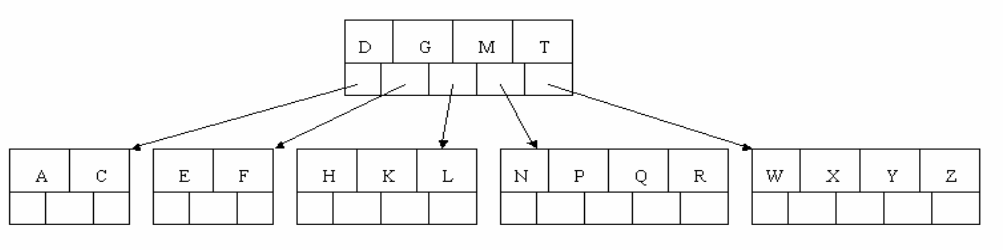
The letters F, W, L, and T are then added without needing any split.



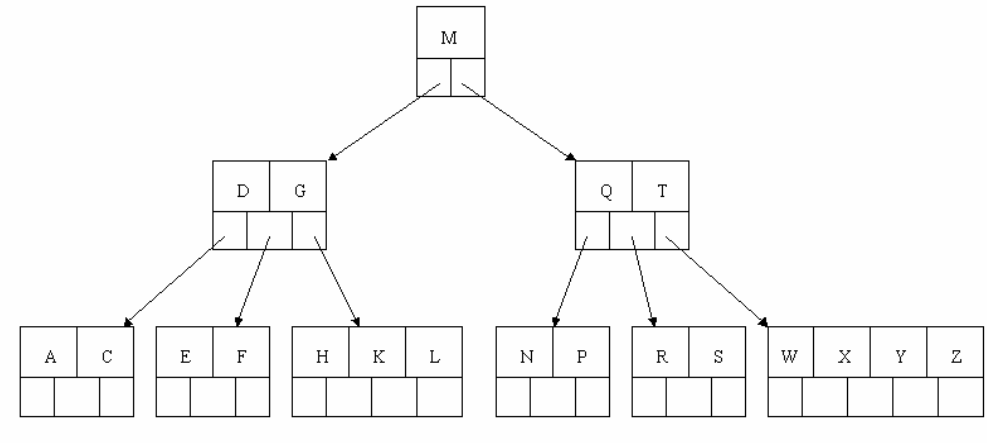
When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.



The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



6.2.3.2 Deletion in B-Tree:

In a B-Tree Deletion takes place at the leaf node

1. Search for the entry to be deleted.
2. If found then continue, else terminate.
3. If it is a leaf simply delete it.
4. If it is an internal node find a successor from its sub-tree and replace it. There are two data items that can be substituted, either the immediate predecessor or immediate successor
5. After deleting if the node has less than the minimum entries (underflow) try Balancing or Combining then continue else terminate

Balancing:

It shifts the data among nodes to reestablish the integrity of the tree. Because it does not change the structure of the tree. We balance a tree by rotating an entry from one sibling to another

through parent. The direction of rotation depends upon the number of siblings in the left or right sub-trees.

.e.g. deletion of R in the next example involves balancing to maintain the integrity of the B Tree

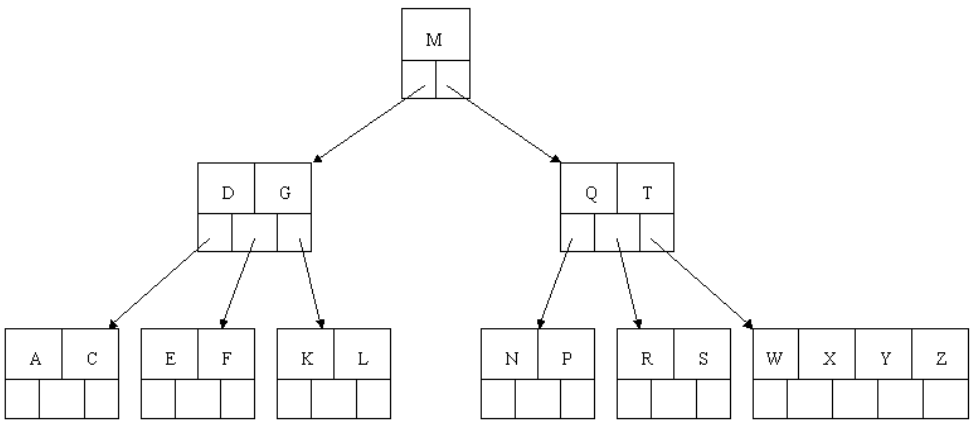
Combining:

Combining joins the data from an under-flowed entry, a minimal sibling and a parent node. The result is one node with maximum entries and an empty node that must be recycled. It makes no difference which sub-tree has under-flowed, we combine all the nodes into left sub-tree. Once this has been done we recycle the right sub-tree.

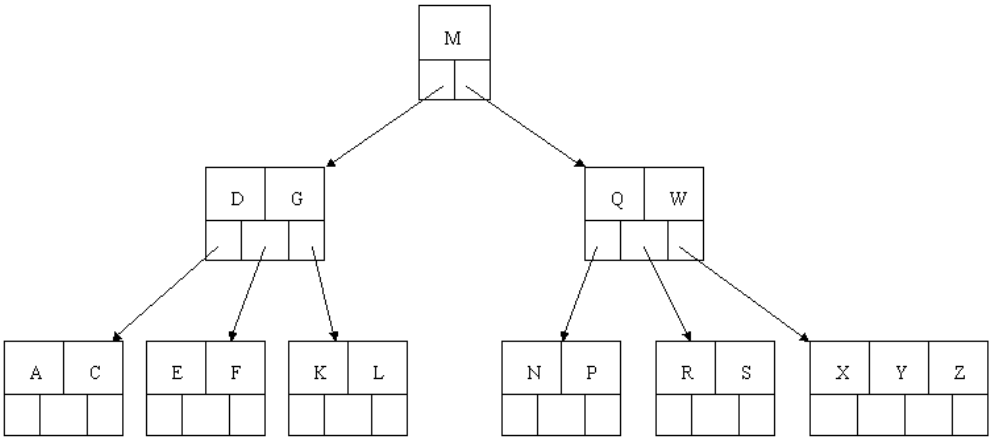
E.g. Deletion of D in the next example results in an underflow and then involves combining of data items.

Example: Deletion in B- Tree

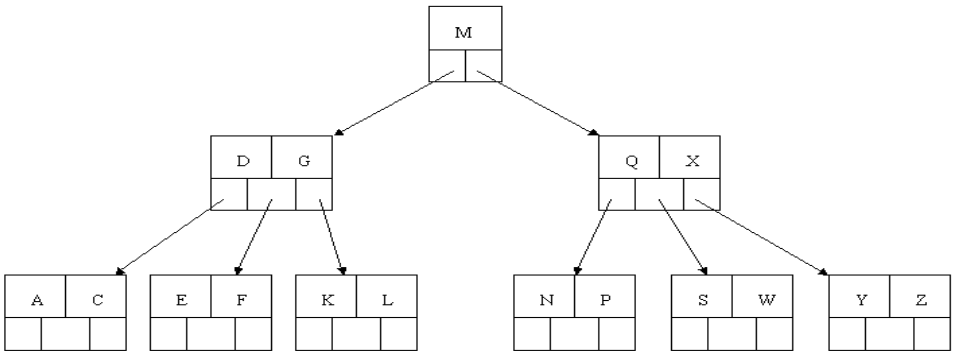
In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been. This gives:



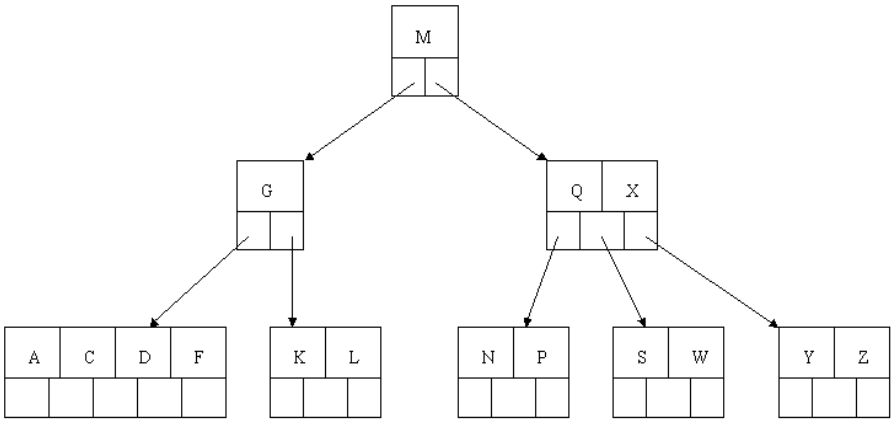
Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method.



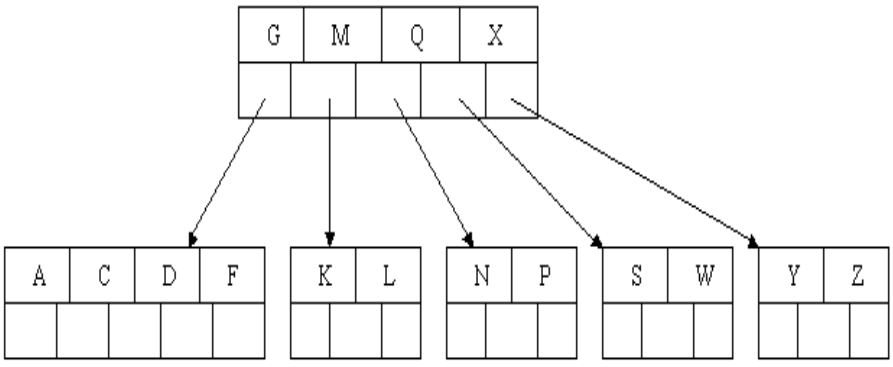
Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)



Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.

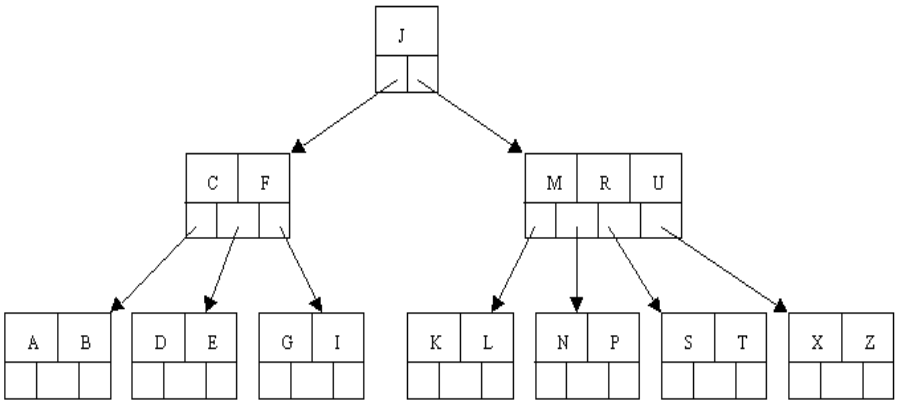


Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M. In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.

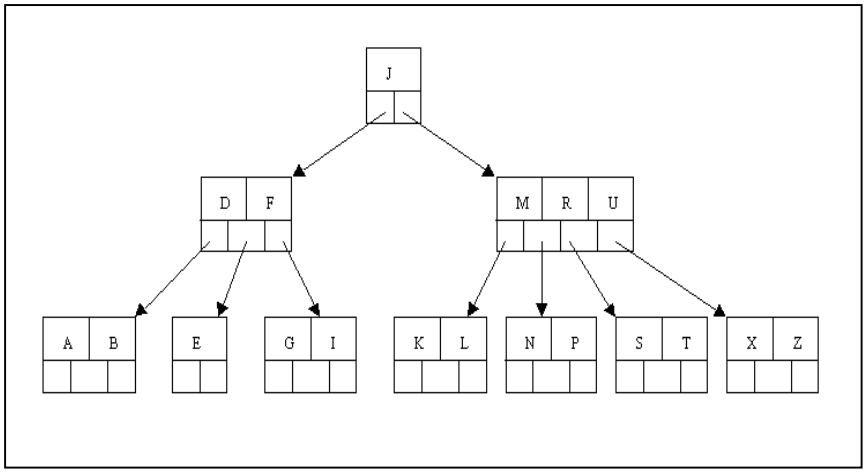


Another Example

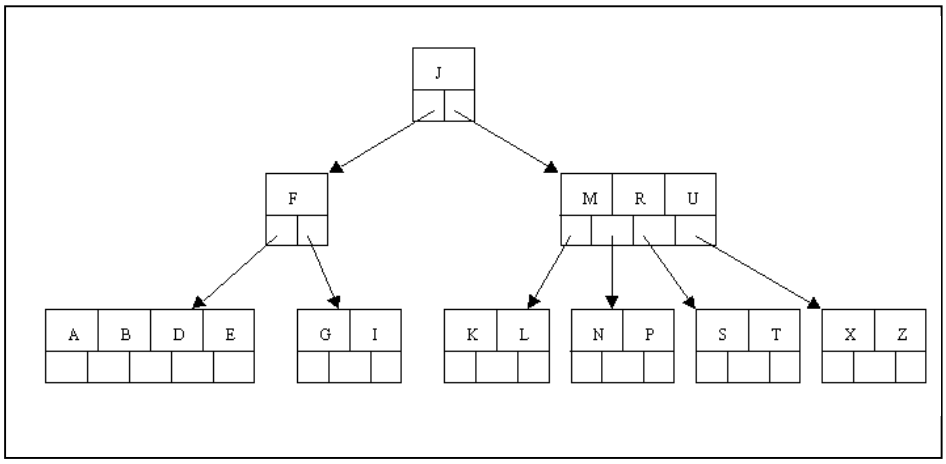
Here is a different B-tree of order 5. Let's try to delete C from it.



We begin by finding the immediate successor, which would be D, and move the D up to replace the C. However, this leaves us with a node with too few keys.



Since neither the sibling to the left or right of the node containing E has an extra key, we must combine the node with one of these two siblings. Let's consolidate with the A B node.



But now the node containing F does not have enough keys. However, its sibling has an extra key. Thus we borrow the M from the sibling, move it up to the parent, and bring the J down to join the F. Note that the K L node gets re-attached to the right of the J.