CISC 220
Midterm Review

Prof. Christopher Rasmussen

http://nameless.cis.udel.edu/class_wiki/index.php/CISC220_F2014

University of Delaware
Fall, 2014

Midterm Details

- Thursday, October 16
- Covers all lectures through Thursday, Oct. 9 class
- Closed book, no notes, no calculators, cell phones, etc.
- Worth 20% of your grade (same as final)
- Question types (see posted 2010 midterm)
  - Data structure ADTs, application examples
  - Definitions, compare/contrast approaches, etc.
  - Write C++ code for some small functions
    - No questions about file I/O or inheritance
  - Calculate, name, rank big-O complexities of various operations
  - Carry out operations and show steps (tree traversals, rotations, etc.)
- Section 010 and 011 versions will be different

Topics Covered

- C++ background, concept of abstract data type (Drozdek, 1.1-1.4 (skip 1.4.5), 1.7-1.8)
- Algorithm analysis (Drozdek, 2.1-2.3, 2.5-2.7)
- Linked lists (Drozdek, 3.1-3.2, 3.7-3.8)
- Stacks & queues (Drozdek, 4.1-4.2, 4.4-4.5)
- Trees (Drozdek, 6.1-6.3, 6.4-6.4.2, 6.5-6.6 (skip 6.6.1, 6.7-6.7.2 (skip 6.7.1))
  - Expression trees (6.12 in 4th ed., 6.10 in 3rd.)

C++ background

- Pointers: & and *
- Arrays (and their relationship to pointers)
- New/delete -- Dynamic vs. static allocation
- Classes vs. objects
  - Constructor, copy constructor, destructor
  - “Shallow” copying and defaults constructor/destructor/assignment vs. overloading
- Operator overloading
- Function and class templates, STL

Linear data structures, lists

- Array-based implementation of stacks, queues
  - Code details for insert(), remove() functions
    - Stack: insert = push, remove = pop
    - Queue: insert = enqueue, remove = dequeue
    - Big-O time complexity
  - Inefficiency of shift operation for queues => circular array
- Singly-linked list implementation of stacks, queues
  - queue needs tail pointer for efficiency
- Doubly-linked list implementation of deques
  - enqueue and dequeue at both ends
  - For which of these operations is SLL inefficient?

Algorithm Analysis

- Counting "worst case" number of primitive operations
  - Be able to do this for functions with nested for loops, conditionals, etc.
- Big-O definition: *f(n)* is O(*g(n)*) if there are positive *c*, n_0 such that *f(n)* <= *g(n)* for *n* >= n_0
  - Simplification of operation count function by dropping constant factors, pulling out biggest term
- Major big-O categories and ordering
  - Constant = O(1)
  - Logarithmic = O(log n)
  - Linear = O(n)
  - Linearithmic = O(n log n)
  - Quadratic = $O(n^2)$
  - Polynomial = $O(n^k)$
  - Exponential = $O(k^n)$

Trees

- Terminology
  - Child, parent, leaf, height, etc.
- Representation
  - General case: nodes + first child/next sibling pointers
  - Binary trees: nodes +left/right child pointers
- Traversals
  - Pre-order, post-order, in-order
- Expression trees
  - Binary trees for representing arithmetic or logical expressions
  - Internal nodes = operators, leaves = operands
  - Which traversal to use for printing, evaluation, etc.
- Binary search trees
  - *Search order* property: keys in left subtree smaller, right keys bigger
  - Details of contains/insert/remove/etc.

AVL trees

- Self-balancing binary trees to keep O(log n) performance
- Definition of *height balance* property
- Balance notation...calculating height difference (+1, 0, -1, etc.) at each node
- Definition of left, right *rotation* at a node
- Analysis of node balance after insert/remove:
  - Which nodes need to have height balance updated?
  - How to decide whether single, double, or no rotation is necessary to fix?
- How many rotations need to be done in the worst case after an insert()? After a remove()?

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 1: Object-Oriented Programming Using C++

Objectives

Looking ahead – in this chapter, we'll consider:

- Abstract Data Types
- Encapsulation
- Inheritance
- Pointers
- Polymorphism

Objectives (continued)

- C++ and Object-Oriented Programming (OOP)
- The Standard Template Library (STL)
- Vectors in the STL
- Data Structures and OOP

Abstract Data Types

- Proper planning is essential to successful implementation of programs
- Typical design methodologies focus on developing models of solutions before implementing them
- These models emphasize structure and function of the algorithms used
- Initially our attention is on **what** needs to be done, not **how** it is done
- So we define program behavior in terms of operations to be performed on data

Abstract Data Types (continued)

- A stack is a **last-in first-out (LIFO)** linear structure where items can only be added and removed from one end
- Operations on this stack ADT might include:
  - PUSH – add an item to the stack
  - POP – remove the item at the top of the stack
  - TOP – return the value of the item at the top of the stack
  - EMPTY – determine if the stack is empty
  - CREATE – create a new empty stack
- Notice these simply describe the things we can do, not how they are done
- These details will be reserved for implementation

Pointers

- A variable defined in a program has two important attributes
  - Its content or value (what it stores)
  - Its location or address (where it is)
- Normally, we access a variable's contents by specifying the variable's name in an operation
- However, it is possible to store a variable's address in another variable
- This new variable is called a pointer; it allows us access to the original variable's value through its address
- A ***pointer*** is a variable whose value is the address of another variable in memory

Pointers (continued)

- Pointers are defined much the same as other variables
  - They have a type; in this case the type of variable they point to
  - They have a user-defined name
  - The name is preceded by an asterisk ("*") to indicate the variable is a pointer
- Given the declarations

```
int i=15,j,*p,*q;
```

  `i` and `j` are integer variables, while `p` and `q` are pointers to integer variables
- If the variable `i` is at memory location 1080, and each variable occupies two bytes, figure 1-1a illustrates this layout

Pointers (continued)

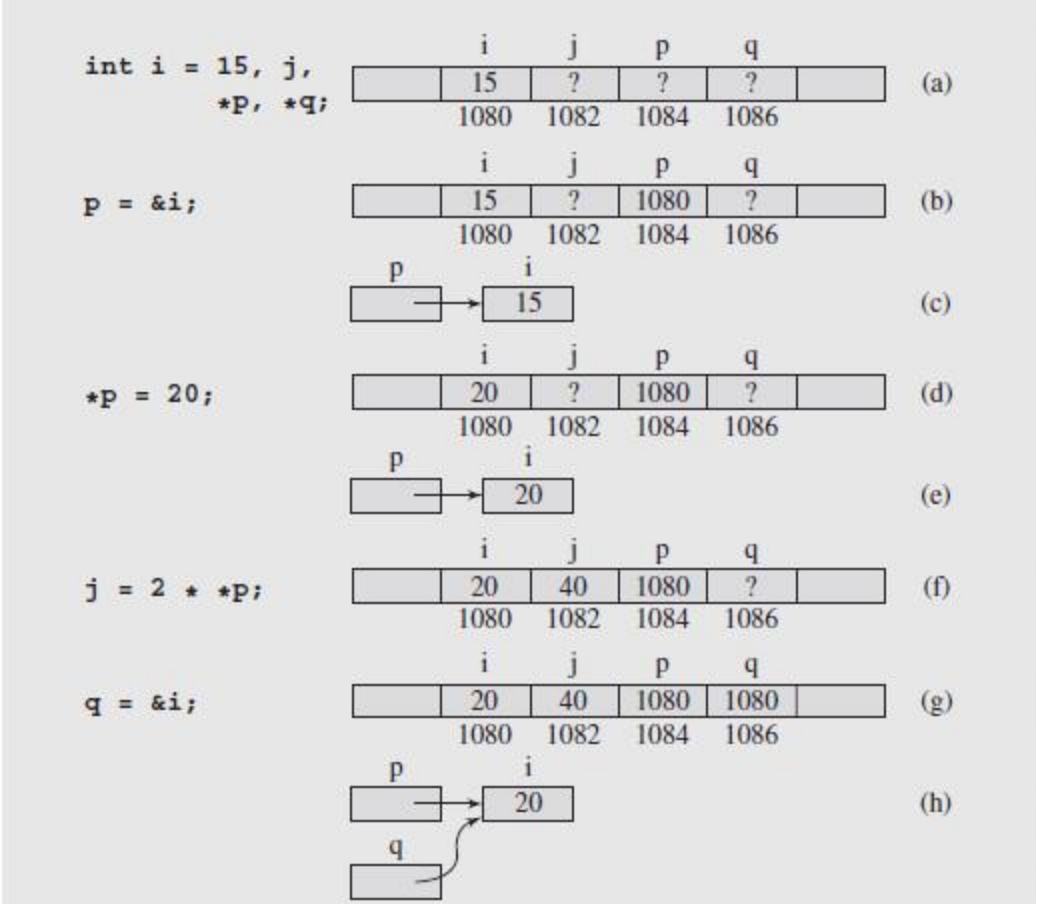


Fig. 1-1 Changes of values after assignments are made using pointer variables; note that (b) and (c) show the same situation, and so do (d) and (e), (g) and (h), (i) and (j), (k) and (l), and (m) and (n)

Pointers (continued)

- To assign a variable's address to a pointer, the **address-of** operator (&) is placed before the variable
- For pointer `p` to point to variable `i`, we write

```
p = &i;
```

- This pointer is then said to **point to** that variable
- This is shown in figure 1-1(b); a common way to draw this relationship is shown in figure 1-1(c)
- To access the value a pointer points to, we have to **dereference** the pointer, using the dereference operator, an asterisk ("*")
- So `*p` refers to the location stored in `p`, the address of `i`

Pointers (continued)

- In addition to variables, pointers can be used to access dynamically created locations
- These are created during runtime using the memory manager
- Two functions are used to handle dynamic memory
  - To allocate memory, `new` is used; it returns the address of the allocated memory, which can be assigned to a pointer

```
p = new int;
```

  - To release the memory pointed at, `delete` is used

```
delete p;
```

- Care must be exercised when using this type of memory operation

Pointers (continued)

- Pointers and Arrays
  - Pointers' ability to access memory locations through their addresses provides us with numerous processing advantages
  - Typically, arrays in C++ are declared before they can be used, known as **static declaration**
  - This means the size of the array must be determined before it is used
  - This is wasteful if the array is too large, or a limitation if the array is too small
  - However, the name of an array is nothing more than a label for the beginning of the array in memory, so it is a pointer

Pointers (continued)

- Pointers and Arrays (continued)
  - For example, an array declared:

```
int temp[7];
```

would appear in memory as:

An array in memory, with the name of the array viewed as a pointer to the first location in the array.

  - In array notation, we access the elements by subscripting: `temp[0]`, `temp[1]`, ... etc.
  - But we can also dereference the pointer to achieve the same results: `*temp` is equivalent to `temp[0]`
  - And we can access additional elements through **pointer arithmetic**

Pointers (continued)

- Pointers and Arrays (continued)
  - In pointer arithmetic, we can add an offset to the base address of the array: `temp + 1`, `temp + 2`, ... etc. and dereference the result
  - So `*(temp + 1)` is the same as `temp[1]`, `*(temp + 2)` is equivalent to `temp[2]`, etc.
  - And as long as we don't try to change the value of `temp`, we can use this alternate approach to access the array's elements
  - Now remember that a pointer can be used in the allocation of memory without a name, through the use of `new`
  - This means that we can also declare an array **dynamically**, via

```
int *p; p = new int[n];
```

  - As long as the value of `n` is known when the declaration is executed, the array can be of arbitrary size

Pointers (continued)

- Pointers and Arrays (continued)
  - Now, a reference to an element of the array is constructed by adding the element's location to `p` and dereferencing, as before
  - And we can assign other pointers to the array by simply declaring them and copying the value of `p` into the new pointer, if needed
  - This also allows us to conveniently dispose of an array when we no longer need it, using:

```
delete[] p;
```

  - The brackets indicate that an array is to be deleted; `p` is the pointer to that array

Pointers (continued)

- Pointers and Copy Constructors
  - A potential problem can arise when copying data from one object to another if one of the data members is a pointer
  - The default behavior is to copy the items member by member
  - Because the value of a pointer is an address, this address is copied to the new object
  - Consequently the new object's pointer points to the same data as the old object's pointer, instead of being distinct
  - To correct this, the user must create a **copy constructor** which will copy not only the pointer, but the object the pointer points to
  - This conflict is illustrated in figure 1-2(a) and (b); the resolution is shown in figure 1-2(c) and (d)

Pointers (continued)

- Pointers and Copy Constructors (continued)

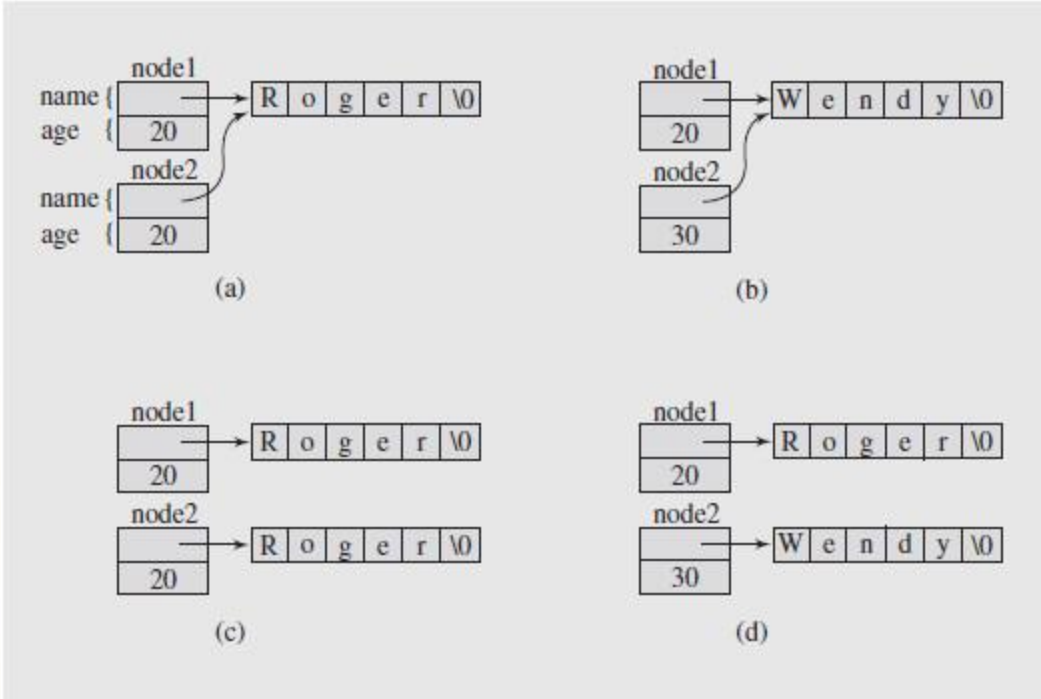


Fig. 1-2 Illustrating the necessity of using a copy constructor for objects with pointer members

Pointers (continued)

- Pointers and Destructors
  - When a local object goes out of scope, the memory associated with it is released
  - Unfortunately, if one of the object members is a pointer, the pointer's memory is released, leaving the object pointed at inaccessible
  - To avoid this memory leak, objects that contain pointers need to have destructors written for them
  - A **destructor** is a code construct that is automatically called when its associated object is deleted
  - It can specify special processing to occur, such as the deletion of pointer-linked memory objects

The Standard Template Library (STL)

- While C++ is a powerful language in its own right, recent additions have added even more capabilities
- Of these, perhaps the most influential is the ***Standard Template Library*** (STL)
- It provides three generic entities: containers, iterators, and algorithms, and a set of classes that overload the function operator called function objects
- It also has a ready-made set of common classes for C++, such as containers and associative arrays
- These can be used with any built-in type and with any user-defined type that supports some elementary operations through templates

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 2: Complexity Analysis

Objectives

Looking ahead – in this chapter, we'll consider:

- Computational and Asymptotic Complexity
- Big-O Notation
- Properties of the Big-O Notation
- Examples of Complexities
- Finding Asymptotic Complexity

Computational and Asymptotic Complexity (continued)

- ***Time complexity*** describes the amount of time an algorithm takes in terms of the amount of input
- ***Space complexity*** describes the amount of memory (space) an algorithm takes in terms of the amount of input
- For both measures, we are interested in the algorithm's ***asymptotic*** complexity
- This asks: when *n* (number of input items) goes to infinity, what happens to the algorithm's performance?

Computational and Asymptotic Complexity (continued)

- To illustrate this, consider $f(n) = n^2 + 100n + \log_{10} n + 1000$
- As the value of n increases, the importance of each term shifts until for large n, only the n^2 term is significant

n	f(n)	n^2		100n		$\log_{10}$n		1,000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.1	100	9.1	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.001	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.00

Fig. 2-1 The growth rate of all terms of function $f(n) = n2 + 100n + \log 10n + 1{,}000$.

Big-O Notation

- The most commonly used notation for asymptotic complexity used is "big-O" notation
- In the previous example we would say $n^2 + 100n + \log_{10} n + 1000 = O(n^2)$ (read "big-oh of n squared")

Definition: Let $f(n)$ and $g(n)$ be functions, where $n \varepsilon Z$ is a positive integer. We write $f(n) = O(g(n))$ if and only if there exists a real number c and positive integer N satisfying $0 \leq f(n) \leq cg(n)$ for all $n \geq N$. (And we say, "f of n is big-oh of g of n.“)

- This means that functions like $n^2 + n$, $4n^2 - n \log n + 12$, $n^2/5 - 100n$, $n \log n$, and so forth are all $O(n^2)$

Possible Problems

- All the notations we've considered focus on comparing algorithms designed to solve the same problem
- We still have to exercise care; it is possible at first glance to eliminate potentially useful candidate functions
- Consider again the definition of big-O

$$f(n) = O(g(n)) \text{ if } 0 \leq f(n) \leq cg(n)$$

- The number of *n*s that violate this is finite and can be reduced by proper choice of *c*
- But if *c* is extremely large, it can cause us to reject a function *g* even if the function itself is promising

Examples of Complexities

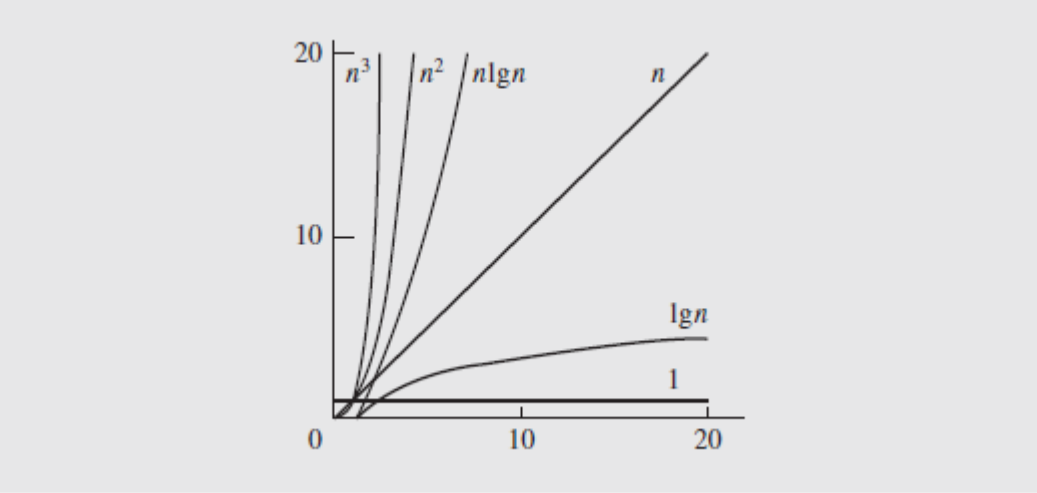
- Since we examine algorithms in terms of their time and space complexity, we can classify them this way, too
- This is illustrated in the next figure

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

Fig. 2.4 Classes of algorithms and their execution times on a computer executing 1 million operations per second (1 sec = 10^6 μsec = 10^3 msec)

Examples of Complexities (continued)

- This relationship can also be expressed graphically:



- This graph, and the previous chart, show that some algorithms have no practical application
- Even with today's supercomputers, cubic order algorithms or higher are impractical for large numbers of elements

Finding Asymptotic Complexity

- As we have seen, asymptotic bounds are used to determine the time and space efficiency of algorithms
- Generally, we are interested in time complexity, which is based on assignments and comparisons in a program
- We'll focus on assignments for the time being
- Consider a simple loop:

```
for (i = sum = 0; i < n; i++)
    sum = sum + a[i]
```

- Two assignments are executed once (`sum = 0` and `i = sum`) during initialization
- In the loop, `sum = sum + a[i]` is executed *n* times

Finding Asymptotic Complexity (continued)

- In addition, the `i++` in the loop header is executed *n* times
- So there are 2 + 2*n* assignments in this loop's execution and it is *O*(*n*)
- Typically, as loops are nested, the complexity grows by a factor of *n*, although this isn't always the case
- Consider

```
for (i = 0; i < n; i++) {
   for (j = 1, sum = a[0]; j <= i; j++)
      sum += a[j];
   cout << "sum for subarray 0 through " << i
           <<" is "<<sum<<end1;
}
```

Finding Asymptotic Complexity (continued)

- The outer loop initializes `i`, then executes *n* times
- During each pass through the loop, the variable `i` is updated, and the inner loop and `cout` statement are executed
- The inner loop initializes `j` and `sum` each time, so the number of assignments so far is 1 + 3n
- The inner loop executes *i* times, where *i* ranges from 1 to *n* – 1, based on the outer loop (when *i* is 0, it doesn't run)
- Each time the inner loop executes, it increments `j`, and assigns a value to `sum`
- So the inner loop executes $\sum_{i=1}^{n-1} 2i = 2(1 + 2 + \ldots + n - 1) = 2n(n - 1)$ assignments

Finding Asymptotic Complexity (continued)

- The total number of assignments is then $1 + 3n + 2n(n - 1)$, which is $O(1) + O(n) + O(n^2) = O(n^2)$
- As mentioned earlier, not all loops increase complexity, so care has to be taken to analyze the processing that takes place
- However, additional complexity can be involved if the number of iterations changes during execution
- This can be the case in some of the more powerful searching and sorting algorithms

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 3: Linked Lists

Objectives

Looking ahead – in this chapter, we'll consider:

- Singly Linked Lists
- Doubly Linked Lists
- Circular Lists
- Skip Lists
- Self-Organizing Lists
- Sparse Tables
- Lists in the Standard Template Library

Introduction

- Arrays are useful in many applications but suffer from two significant limitations
  - The size of the array must be known at the time the code is compiled
  - The elements of the array are the same distance apart in memory, requiring potentially extensive shifting when inserting a new element
- This can be overcome by using ***linked lists***, collections of independent memory locations (***nodes***) that store data and links to other nodes
- Moving between the nodes is accomplished by following the links, which are the addresses of the nodes
- There are numerous ways to implement linked lists, but the most common utilizes pointers, providing great flexibility

Singly Linked Lists

- If a node contains a pointer to another node, we can string together any number of nodes, and need only a single variable to access the sequence
- In its simplest form, each node is composed of a datum and a link (the address) to the next node in the sequence
- This is called a ***singly linked list***, illustrated in figure 3.1
- Notice the single variable *p* used to access the entire list
- Also note the last node in the list has a null pointer (\ )

Singly Linked Lists (continued)

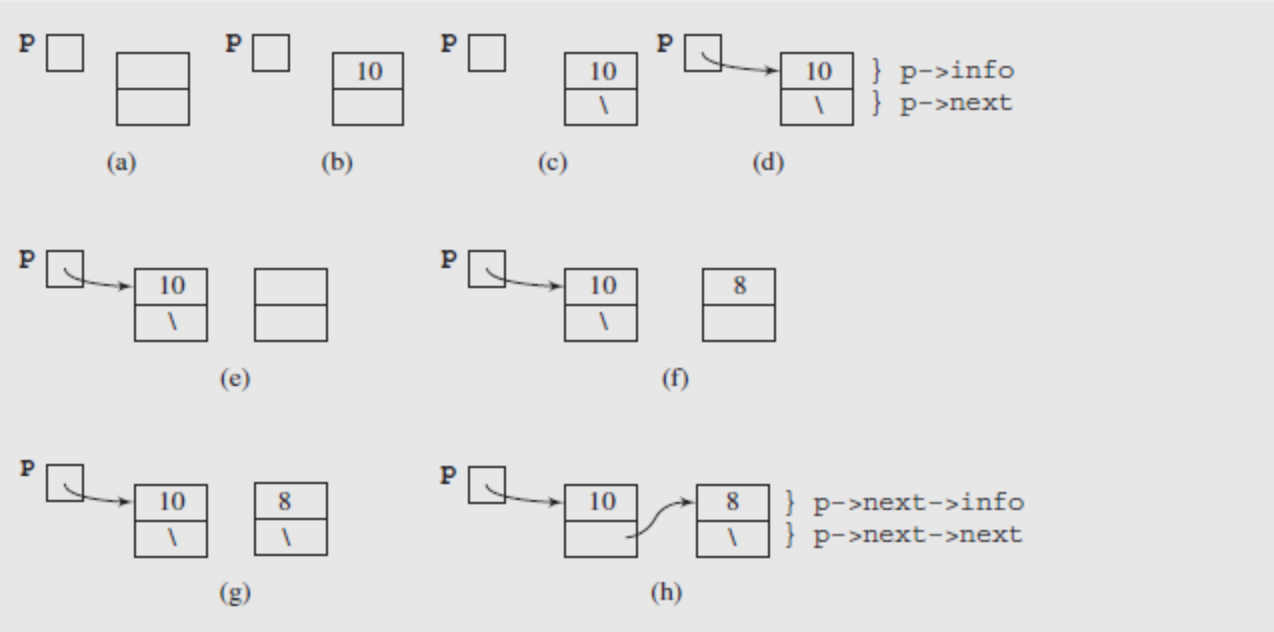


Fig. 3.1 A singly linked list

- The nodes in this list are objects created from the following class definition

Singly Linked Lists (continued)

```
class IntSLLNode {
public:
    IntSLLNode() {
    next = 0;
    }
    IntSLLNode(int i, IntSLLNode *in = 0){
    info = i; next = in;
    }
    int info;
    IntSLLNode *next;
}
```

- As can be seen here and in the previous figure, a node consists of two data members, `info` and `next`

Singly Linked Lists (continued)

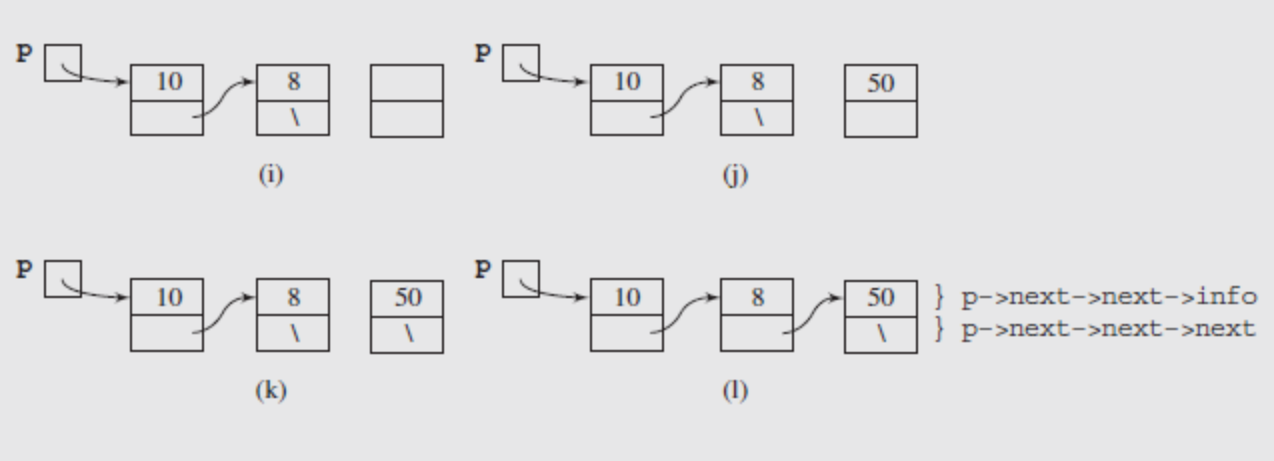


Fig. 3.1 (continued) A singly linked list

- This example illustrates a disadvantage of single-linked lists: the longer the list, the longer the chain of `next` pointers that need to be followed to a given node
- This reduces flexibility, and is prone to errors
- An alternative is to use an additional pointer to the end of the list

Singly Linked Lists (continued)

- Deletion
  - The operation of deleting a node consists of returning the value stored in the node and releasing the memory occupied by the node
  - Again, we can consider operations at the beginning and end of the list
  - To delete at the beginning of the list, we first retrieve the value stored in the first node (`head → info`)
  - Then we can use a temporary pointer to point to the node, and set `head` to point to `head → next`
  - Finally, the former first node can be deleted, releasing its memory
  - These operations are illustrated in figure 3.6(a) – (c)

Singly Linked Lists (continued)

- Deletion (continued)

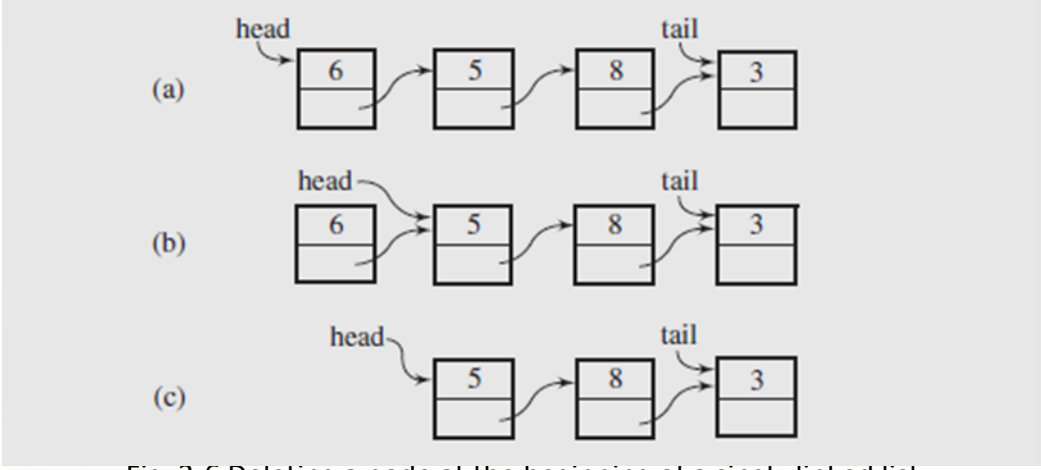


Fig. 3-6 Deleting a node at the beginning of a singly linked list

  - Two special cases exist when carrying out this deletion
  - The first arises when the list is empty, in which case the caller must be notified of the action to take
  - The second occurs when a single node is in the list, requiring that `head` and `tail` be set to null to indicate the list is now empty

Singly Linked Lists (continued)

- Searching
  - The purpose of a search is to scan a linked list to find a particular data member
  - No modification is made to the list, so this can be done easily using a single temporary pointer
  - We simply traverse the list until the `info` member of the node `tmp` points to matches the target, or `tmp → next` is null
  - If the latter case occurs, we have reached the end of the list and the search fails

Doubly Linked Lists

- The difficulty in deleting a node from the end of a singly linked list points out one major limitation of that structure
- We continually have to scan to the node just before the end in order to delete correctly
- If the nature of processing requires frequent deletions of that type, this significantly slows down operations
- To address this problem, we can redefine the node structure and add a second pointer that points to the previous node
- Lists constructed from these nodes are called ***doubly linked lists***, one of which is shown in figure 3-9

Doubly Linked Lists (continued)

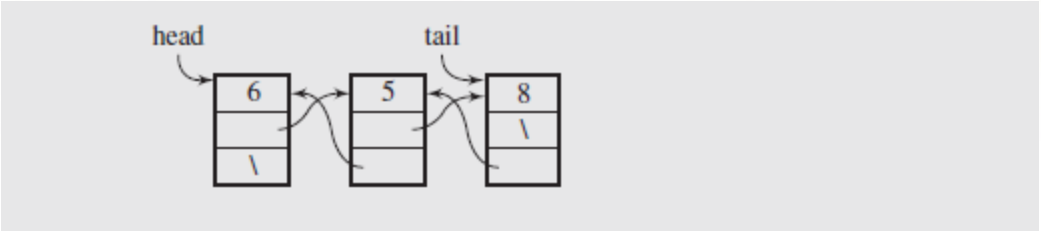


Fig. 3-9 A doubly linked list

- The code that implements and manipulates this is shown in part in figure 3-10 (pages 91 and 92)
- The methods that manipulate these types of lists are slightly more complicated than their singly linked counterparts
- However, the process is still straightforward as long as one keeps track of the pointers and their relationships

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 4: Stacks and Queues

Objectives

Looking ahead – in this chapter, we'll consider

- Stacks
- Queues
- Priority Queues
- Stacks in the Standard Template Library
- Queues in the Standard Template Library
- Priority Queues in the Standard Template Library
- Deques in the Standard Template Library

Stacks

- A ***stack*** is a restricted access linear data structure
- It can only be accessed at one of its ends for adding and removing data elements
- A classic analogy is of a stack of trays in a cafeteria; trays are removed from the top and placed back on the top
- So the very first tray in the pile is the last one to be removed
- For this reason, stacks are also known as ***last-in first-out (LIFO)*** structures
- We can only remove items that are available, and can't add more items if there is no room, so we can define the stack in terms of operations that change it or report its status

Stacks (continued)

- These operations are:
  - ***clear( )***: clears the stack
  - ***isEmpty( )***: determines if the stack is empty
  - ***push(el)***: pushes the data item ***el*** onto the top of the stack
  - ***pop( )***: removes the top element from the stack
  - ***topEl( )***: returns the value of the top element of the stack without removing it
- A series of pushes and pops is shown in figure 4.1

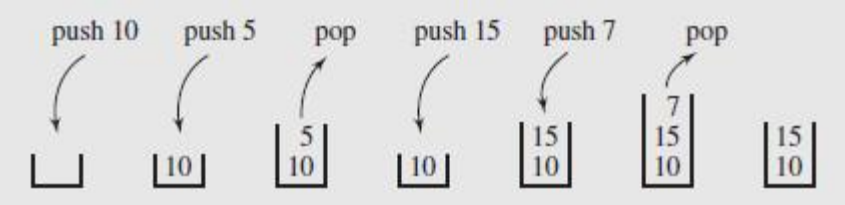


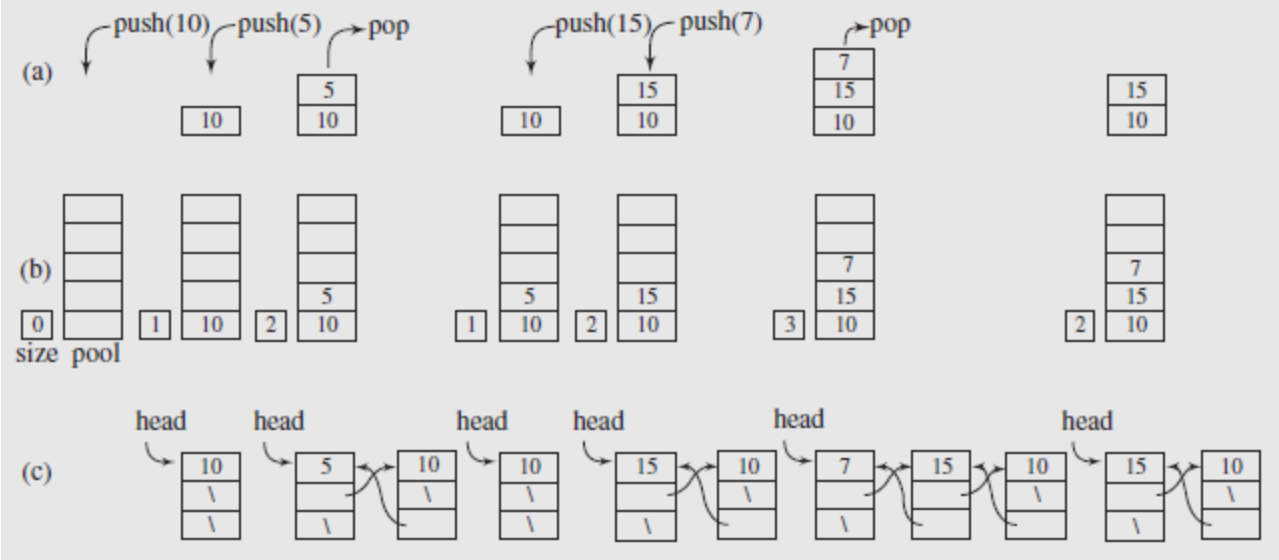
Fig. 4.1 A series of operations executed on a stack

Stacks (continued)

- Stacks are particularly useful in situations where data have to be stored and processed in reverse order
- There are numerous applications of this
  - Evaluating expressions and parsing syntax
  - Balancing delimiters in program code
  - Converting numbers between bases
  - Processing financial data
  - Backtracking algorithms
- An example of delimiter processing is shown in figure 4.2 (page 134), a second example deals with adding large numbers (figure 4.3, page 135)

Stacks (continued)

- Let's now turn our attention to implementing the stack ADT
- One possible implementation uses a vector/array ((b) below), and another uses a linked list ((c) below)



Queues

- A ***queue***, like a stack, is a restricted access linear data structure
- Unlike a stack, both ends are involved, with additions restricted to one end (the ***rear***) and deletions to the other (the ***front***)
- Since an item added to the queue must migrate from the rear to the front before it is removed, items are removed in the order they are added
- For this reason, queues are also known as ***first-in first-out (FIFO)*** structures

Queues (continued)

- The functions of a queue are similar to those of a stack
- Typically, the following methods are implemented
  - ***clear( )***: clears the queue
  - ***isEmpty( )***: determines if the queue is empty
  - ***enqueue(el)***: adds the data item ***el*** to the end of the queue
  - ***dequeue( )***: removes the element from the front of the queue
  - ***firstEl( )***: returns the value of the first element of the queue without removing it
- A series of enqueues and dequeues is shown in figure 4.7
- Note that this time both ends of the structure must be managed

Queues (continued)

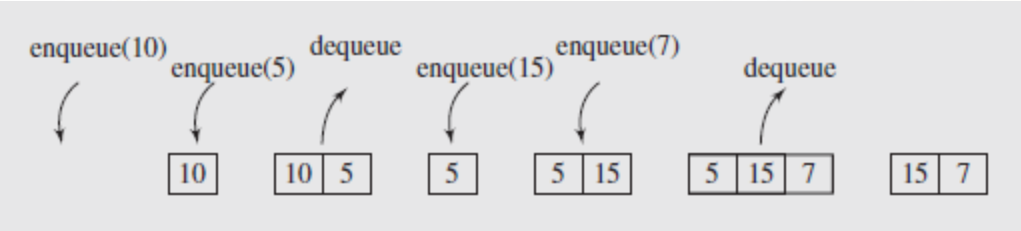


Fig. 4.7 A series of operations executed on a queue

- One way a queue may be implemented utilizes an array, although care must be exercised
- In particular, as items are removed from the queue, spaces open up in the front of the array, which should not be wasted
- So items may be added to the “end” of the queue at the beginning of the array
- This treats the array as though it were circular, shown in figure 4.8c

Queues (continued)

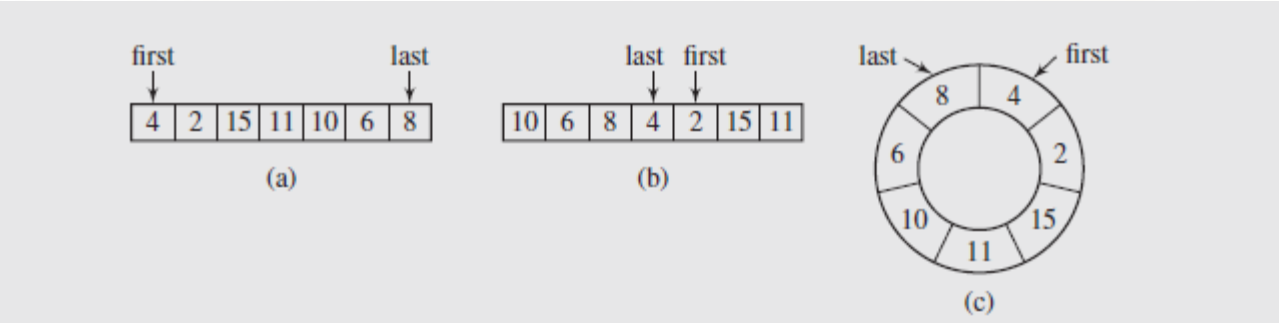


Fig. 4.8 (a–b) Two possible configurations in an array implementation of a queue when the queue is full; (c) the same queue viewed as a circular array

- As the circular array illustrates, the queue is full if the first and last elements are adjacent
- However, based on the actual array, this can occur in two situations, shown in figure 4.8(a) and figure 4.8(b):
  - The first element is in the first location, and the last element in the last
  - The first element is immediately after the last element

Queues (continued)

- This also means that the *enqueue( )* and *dequeue( )* operations have to deal with wrapping around the array
- Adding an element may require placing it at the beginning of the array (figure 4.8d) or after the last element (figure 4.8e) if there is room, even though the circular array doesn't distinguish these (figure 4.8f)

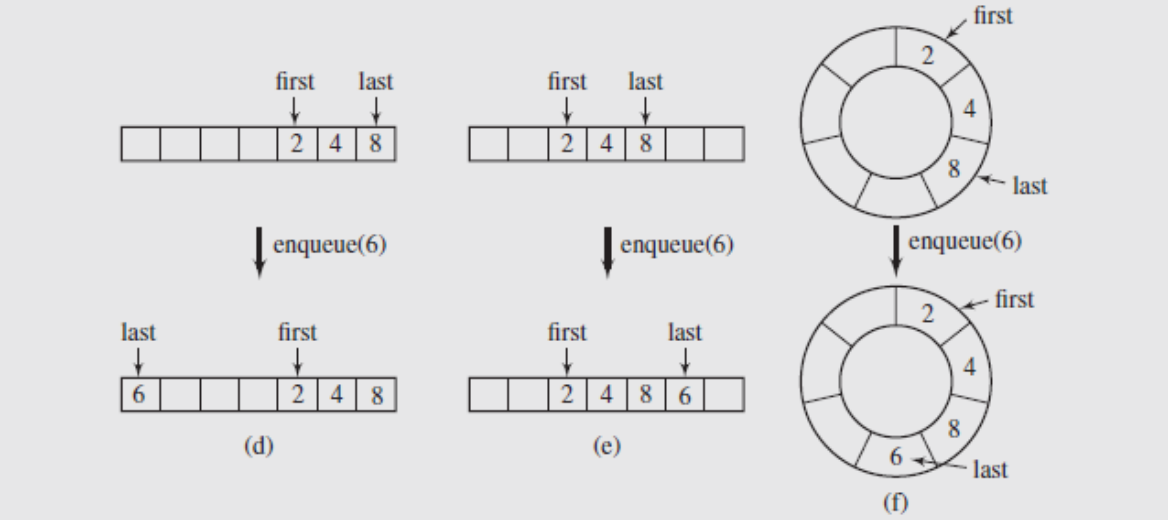


Fig. 4.8 (continued) (f) Enqueuing number 6 to a queue storing 2, 4, and 8; (d–e) the same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle

Queues (continued)

- Figure 4.9 (pages 142 and 143) shows some possible implementations of methods to work on such a queue
- A second, more flexible implementation of a queue is as a doubly linked list, either coded directly or as an STL `list`
- This is shown in figure 4.10
- Figure 4.11 shows the same enqueue and dequeue operations that were illustrated in figure 4.7
- They also indicate the changes that are needed in the queue when implemented as an array (figure 4.11b) or linked list (figure 4.11c)

Queues (continued)

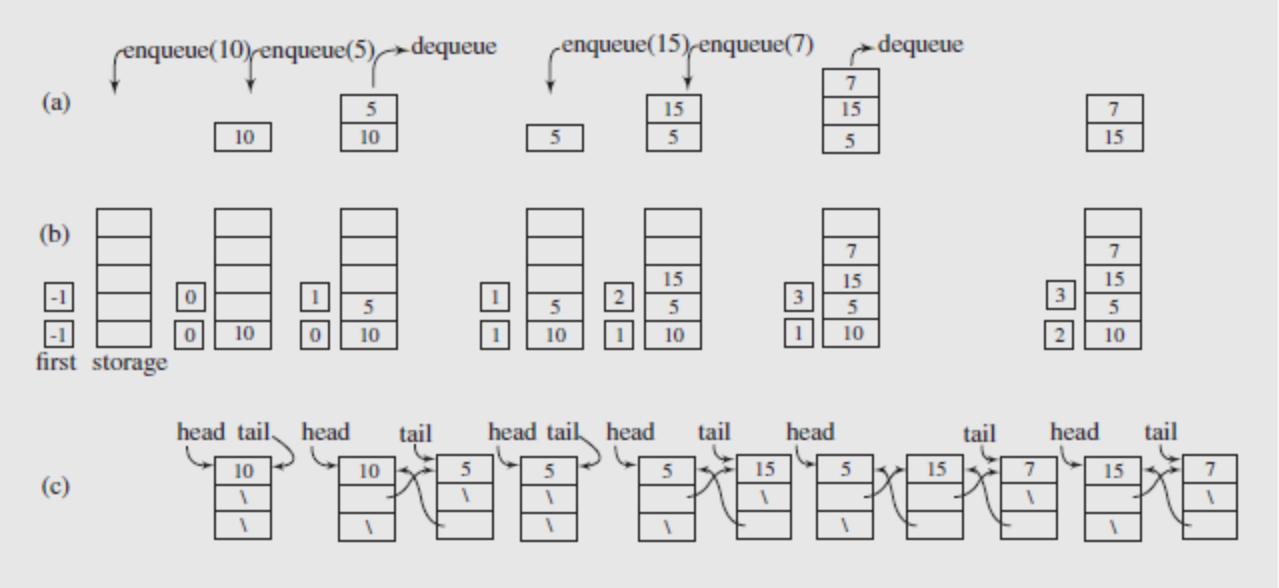


Fig. 4.11 A series of operations executed on (a) an abstract queue and the queue implemented (b) with an array and (c) with a linked list

Stacks in the Standard Template Library

- The STL implements the stack as a container adaptor
- This is not a new container, merely an adaptation of an existing one to make the stack behave in a specific way
- The `deque` is the default container, but lists and vectors can also be used

```
stack<int> stack1;                  // deque by default
stack<int,vector<int>> stack2;// vector
stack<int,list<int>> stack3;  // list
```

- The stack container methods are shown in figure 4.14
- Notice that `pop()` does not return a value; to implement this, `pop()` must be combined with `top()`

Stacks in the Standard Template Library (continued)

Member Function	Operation
`bool empty() const`	Return `true` if the stack includes no element and `false` otherwise.
`void pop()`	Remove the top element of the stack.
`void push(const T& el)`	Insert `el` at the top of the stack.
`size_type size() const`	Return the number of elements on the stack.
`stack()`	Create an empty stack.
`T& top()`	Return the top element on the stack.
`const T& top() const`	Return the top element on the stack.

Fig. 4.14 A list of stack member functions

Queues in the Standard Template Library

- By default, the queue in the STL is implemented as an adapted `deque`, although a `list` can be used instead
- Figure 4.15 shows the methods for a queue

Member Function	Operation
`T& back()`	Return the last element in the queue.
`const T& back() const`	Return the last element in the queue.
`bool empty() const`	Return `true` if the queue includes no element and `false` otherwise.
`T& front()`	Return the first element in the queue.
`const T& front() const`	Return the first element in the queue.
`void pop()`	Remove the first element in the queue.
`void push(const T& el)`	Insert `el` at the end of the queue.
`queue()`	Create an empty queue.
`size_type size() const`	Return the number of elements in the queue.

Fig. 4.15 A list of queue member functions

- These methods are illustrated in the program in figure 4.16

DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 6: Binary Trees

Objectives

Looking ahead – in this chapter, we'll consider

- Trees, Binary Trees, and Binary Search Trees
- Implementing Binary Trees
- Searching a Binary Search Tree
- Tree Traversal
- Insertion/Deletion
- Balancing a Tree
- Polish Notation and Expression Trees

Trees, Binary Trees, and Binary Search Trees

- While linked lists, stacks, and queues are useful data structures, they do have limitations
  - Linked lists are linear in form and cannot reflect hierarchically organized data
  - Stacks and queues are one-dimensional structures and have limited expressiveness
- To overcome these limitations, we'll consider a new data structure, the ***tree***
- Trees consist of two components, ***nodes*** and ***arcs*** (or ***edges***)
- Trees are drawn with the ***root*** at the top, and "grow" down
  - The ***leaves*** of the tree (also called ***terminal nodes***) are at the bottom of the tree

Trees, Binary Trees, and Binary Search Trees (continued)

- Trees can be defined recursively as follows:
  1. A tree with no nodes or arcs (an empty structure) is an empty tree
  2. If we have a set $t_1 \cdots t_k$ of disjoint trees, the tree whose root has the roots of $t_1 \cdots t_k$ as its children is a tree
  3. Only structures generated by rules 1 and 2 are trees
- Every node in the tree must be accessible from the root through a unique sequence of arcs, called a ***path***
- The number of arcs in the path is the path's ***length***
- A node's ***level*** is the length of the path to that node, plus 1

Trees, Binary Trees, and Binary Search Trees (continued)

- The maximum level of a node in a tree is the tree's ***height***
- An empty tree has height 0, and a tree of height 1 consists of a single node which is both the tree's root and leaf
- The level of a node must be between 1 and the tree's height

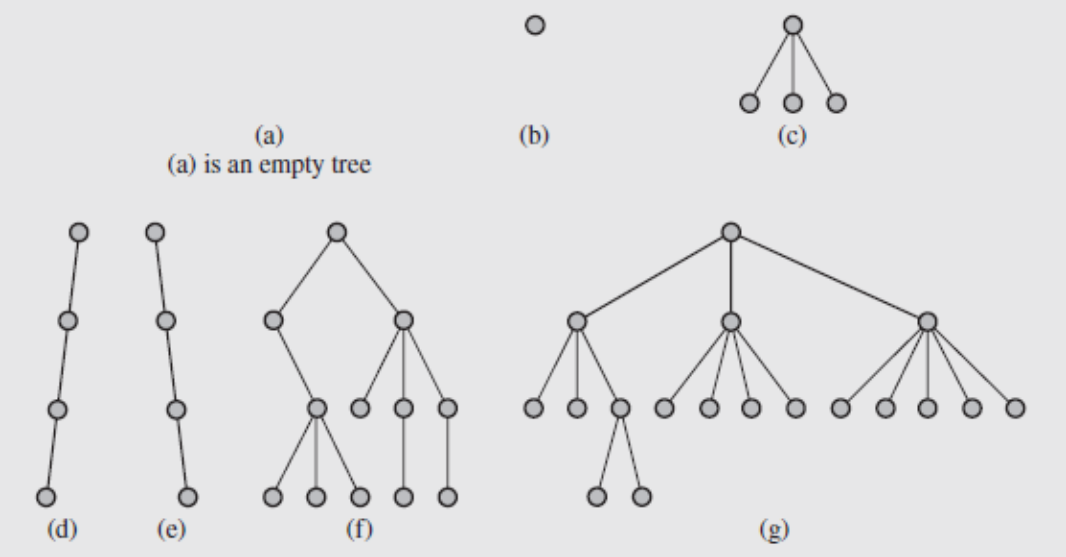


Fig. 6.1 Some examples of trees

Trees, Binary Trees, and Binary Search Trees (continued)

- A binary tree is a tree where each node has only two children, designated the ***left child*** and the ***right child***
- These children can be empty; Figure 6.4 shows examples of binary trees

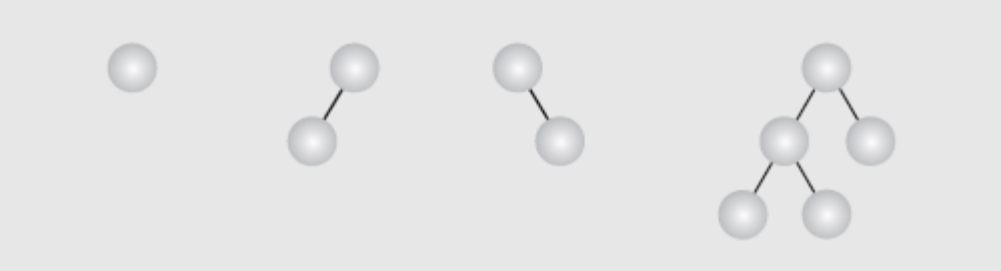


Fig. 6.4 Examples of binary trees

- An important attribute of binary trees is the number of leaves
- This is useful in assessing efficiency of algorithms

Tree Traversal

- ***Tree traversal*** is the process of visiting each node in a tree data structure exactly one time
- This definition only specifies that each node is visited, but does not indicate the order of the process
- Hence, there are numerous possible traversals; in a tree of *n* nodes there are *n*! traversals
- Two especially useful traversals are ***depth-first traversals*** and ***breadth-first traversals***

Tree Traversal (continued)

- Depth-First Traversal
  - Depth-first traversal proceeds by following left- (or right-) hand branches as far as possible
  - The algorithm then backtracks to the most recent fork and takes the right- (or left-) hand branch to the next node
  - It then follows branches to the left (or right) again as far as possible
  - This process continues until all nodes have been visited
  - While this process is straightforward, it doesn't indicate at what point the nodes are visited; there are variations that can be used
  - We are interested in three activities: traversing to the left, traversing to the right, and visiting a node
    - These activities are labeled L, R, and V, for ease of representation

Tree Traversal (continued)

- Depth-First Traversal (continued)
  - Based on earlier discussions, we want to perform the traversal in an orderly manner, so there are six possible arrangements:
    - VLR, VRL, LVR, LRV, RVL, and RLV
  - Generally, we follow the convention of traversing from left to right, which narrows this down to three traversals:
    - VLR – known as ***preorder traversal***
    - LVR – known as ***inorder traversal***
    - LRV – known as ***postorder traversal***
  - These can be implemented straightforwardly, as seen in Figure 6.11

Tree Traversal (continued)

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p) {
     if (p != 0) {
         inorder(p->left);
         visit(p);
         inorder(p->right);
     }
}
```

```
template<class T>
void BST<T>::postorder(BSTNode<T>* p) {
    if (p != 0) {
        postorder(p->left);
        postorder(p->right);
        visit(p);
    }
}
```

```
template<class T>
void BST<T>::preorder(BSTNode<T> *p) {
    if (p != 0) {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}
```

Fig. 6.11 Depth-first traversal implementations

Tree Traversal (continued)

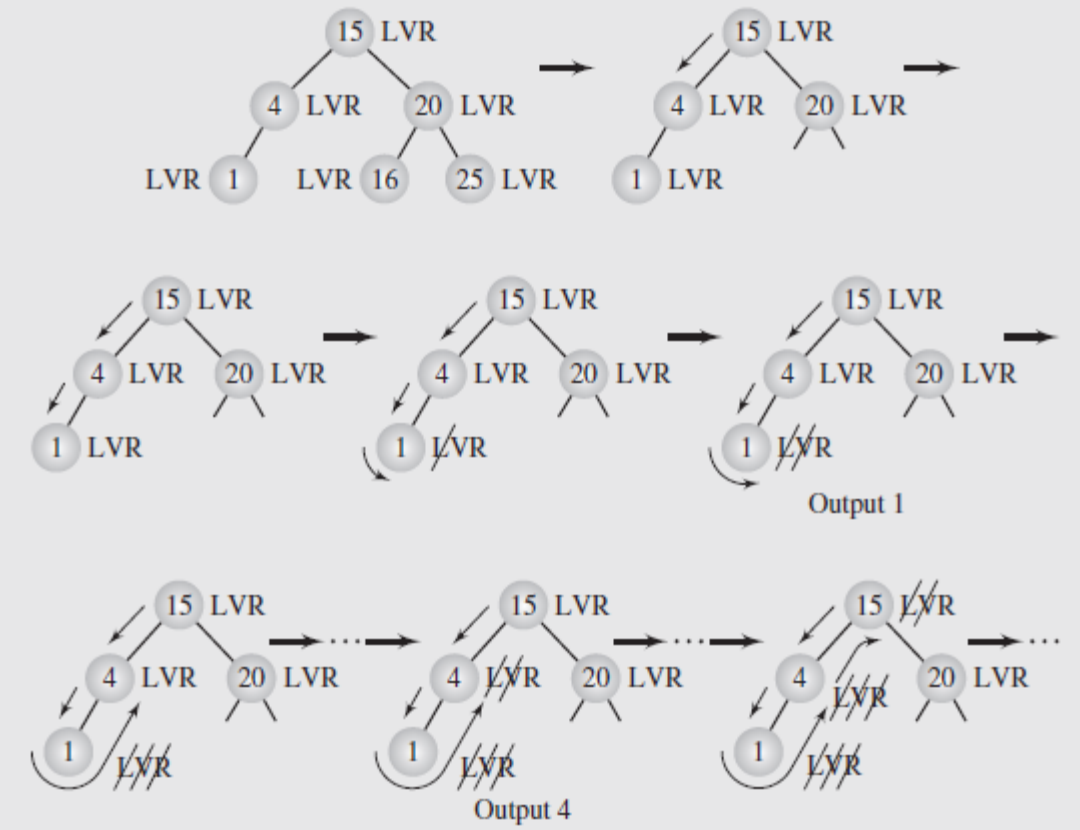


Fig. 6.13 Details of several of the first steps of inorder traversal

Expression Trees

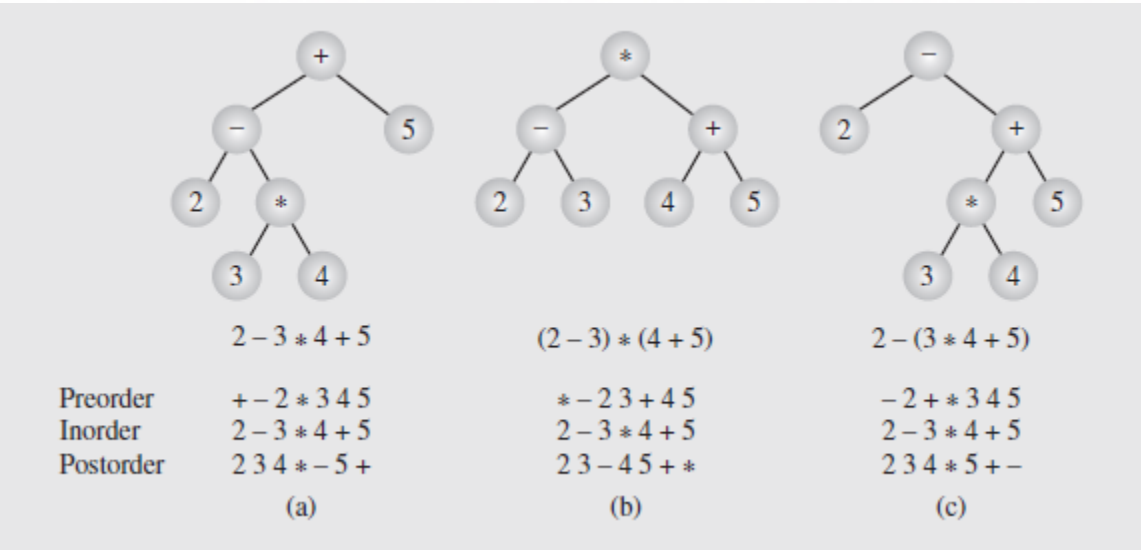


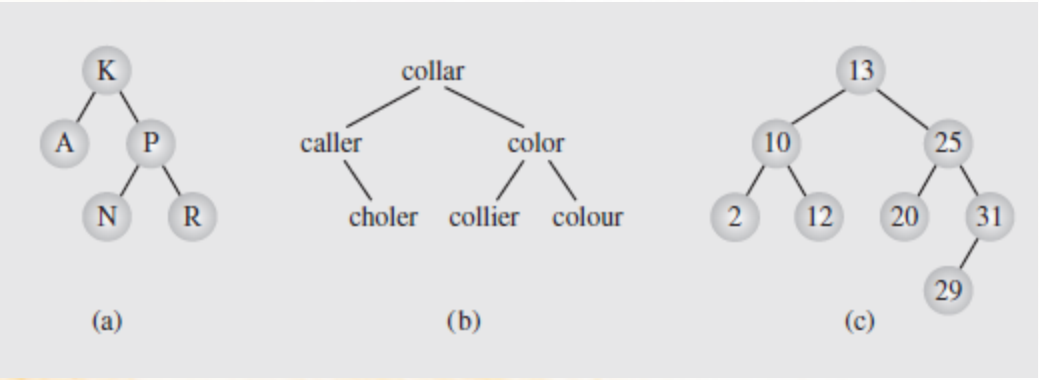
Fig. 6.64 Examples of three expression trees and results of their traversals

Expression Trees (continued)

- Because of the usefulness and importance of these traversals, the results they produce are given names to distinguish them
- Preorder traversals produce ***prefix notation***, where the operator precedes the operands it works on, such as in LISP
- Postorder traversals generate ***postfix notation***, where the operator follows the operands it works on, such as in Forth
- Inorder traversals create ***infix notation***, where the operator is in between its operands
- It is the infix notation form that we are most familiar with in reading and creating expressions

Trees, Binary Trees, and Binary Search Trees (continued)

- In a ***binary search tree*** (or ***ordered binary tree***):
  - Values ("keys") stored in the left subtree of a given node are < the value stored in that node
  - Values stored in the right subtree are > the value stored in that node
  - The values stored are considered unique; attempts to store duplicate values can be treated as an error or a counter variable can be incremented
  - The meanings of the expressions "less than" and "greater than" will depend on the types of values stored



Searching a Binary Search Tree

- Locating a specific value in a binary tree is easy:

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
    while (p != 0)
        if (el == p->el)
            return &p->el;
        else if (el < p->el)
            p = p->left;
        else p = p->right;
    return 0;
}
```

Fig. 6.9 A function for searching a binary search tree

- For each node, compare the value to the target value; if they match, the search is done
- If the target is smaller, we branch to the left subtree; if larger, we branch to the right
- If at any point we cannot proceed further, then the search has failed and the target isn't in the tree

Searching a Binary Search Tree (continued)

- The number of comparisons performed during the search determines the complexity of the search
- This in turn depends on the number of nodes encountered on the path from the root to the target node
- So the complexity is the length of the path plus 1, and is influenced by the shape of the tree and location of the target
- Searching in a binary tree is quite efficient, even if it isn't balanced
- However, this only holds for randomly created trees, as those that are highly unbalanced or elongated and resemble linear linked lists approach sequential search times

Insertion

- Searching a binary tree does not modify the tree
- Traversals may temporarily modify the tree, but it is usually left in its original form when the traversal is done
- Operations like insertions, deletions, modifying values, merging trees, and balancing trees do alter the tree structure
- We'll look at how insertions are managed in binary search trees first
- In order to insert a new node in a binary tree, we have to be at a node with a vacant left or right child
- This is performed in the same way as searching:
  - Compare the value of the node to be inserted to the current node
  - If the value to be inserted is smaller, follow the left subtree; if it is larger, follow the right subtree
  - If the branch we are to follow is empty, we stop the search and insert the new node as that child

Insertion (continued)

- This process is shown in Figure 6.22; the code to implement this algorithm shown in Figure 6.23

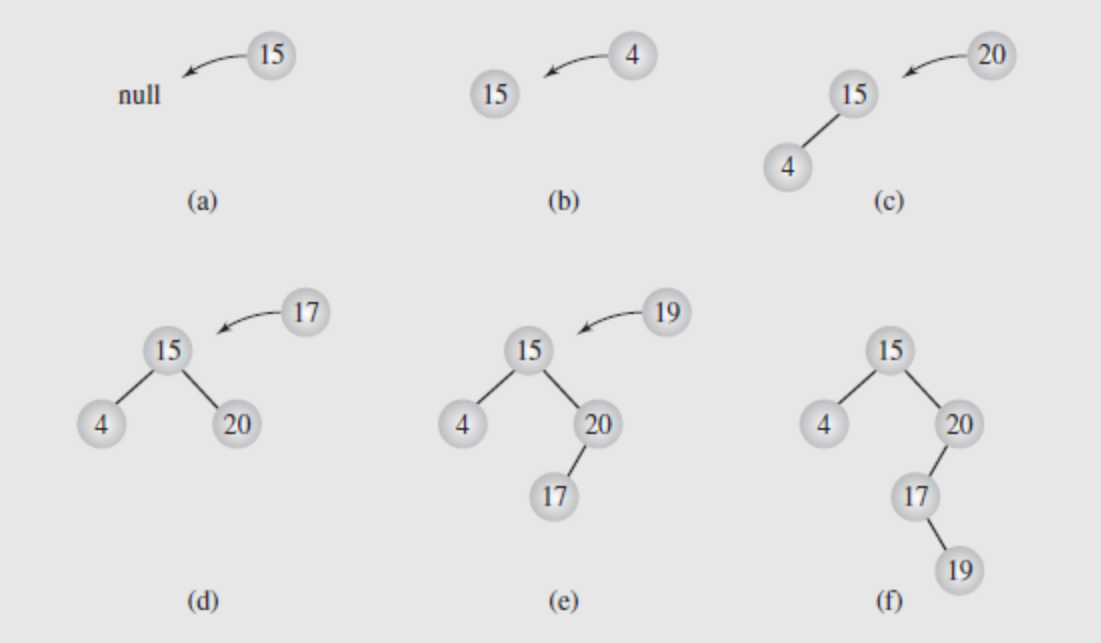


Fig. 6.22 Inserting nodes into binary search trees

Deletion

- Deletion is another operation essential to maintaining a binary search tree
- This can be a complex operation depending on the placement of the node to be deleted in the tree
- The more children a node has, the more complex the deletion process
- This implies three cases of deletion that need to be handled:
  - The node is a leaf; this is the easiest case, because all that needs to be done is to set the parent link to null and delete the node (Figure 6.26)
  - The node has one child; also easy, as we set the parent's pointer to the node to point to the node's child (Figure 6.27)

Deletion (continued)

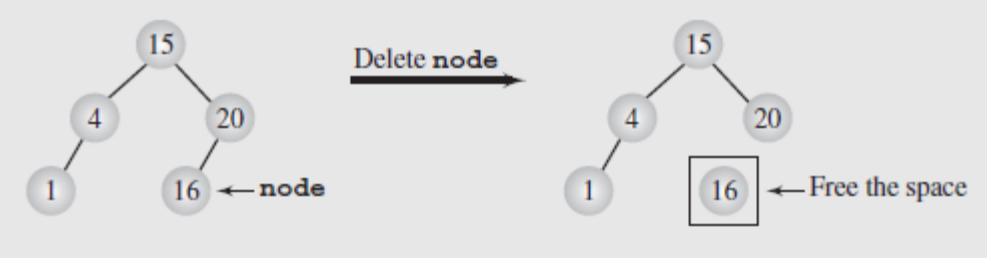


Fig. 6.26 Deleting a leaf

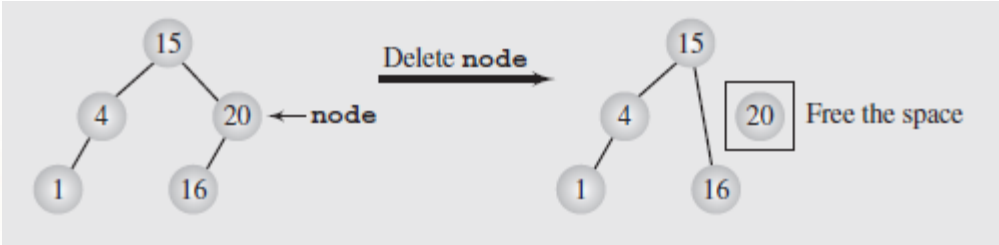


Fig. 6.27 Deleting a node with one child

– The third case, and most difficult to handle, is when the node has two children, as there is no one-step process; we'll consider two options

Deletion (continued)

- Deletion by Copying
  - Another approach to handling deleting is called ***deletion by copying*** and was proposed by Thomas Hibbard and Donald Knuth in the 1960s
  - We locate the node's predecessor by searching for the rightmost node in the left subtree (the "predecessor" key) OR the leftmost node in the right subtree (the "successor" key)
  - The key of this node replaces the key of the node to be deleted
  - We then recall the two simple cases of deletion: if the rightmost node was a leaf, we delete it; if it has one child, we set the parent's pointer to the node to point to the node's child
  - This way, we delete a key k_1 by overwriting it by a key k_2 and then deleting the node holding k_2

Deletion (continued)

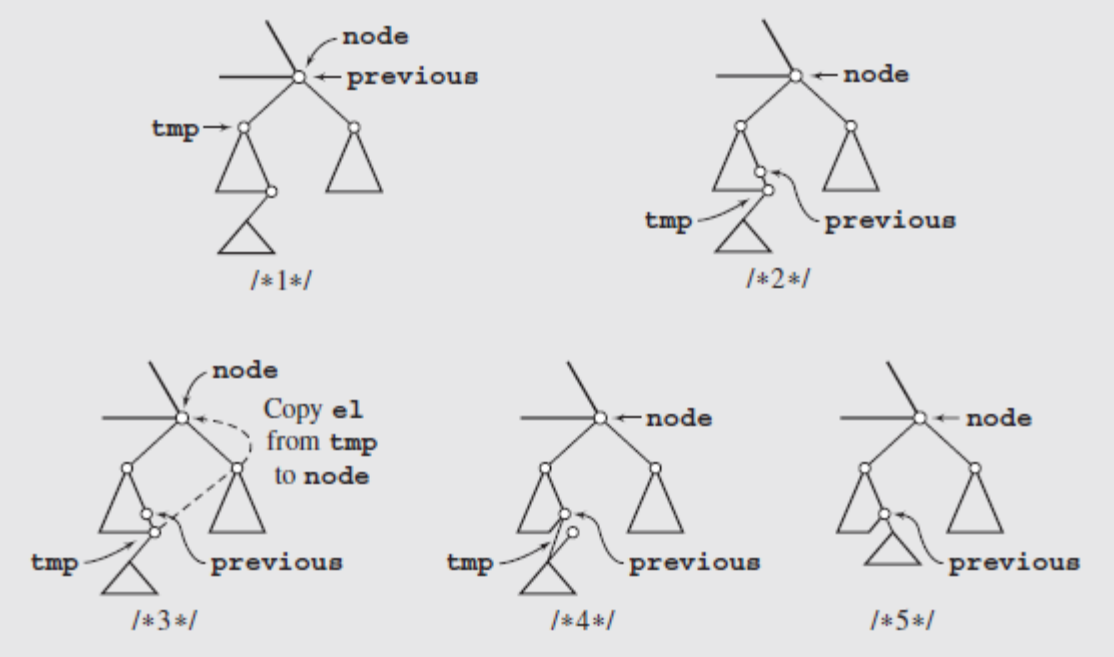


Figure 6.33 Deleting by copying

Deletion (continued)

- Deletion by Copying (continued)
  - Eventually the tree becomes unbalanced to the right, and the right subtree is bushier and larger than the left
  - A simple improvement can make this symmetric; we alternately delete the node's predecessor from the left subtree and its successor from the right subtree
  - This provides significant improvements; however the analysis has proven to be extremely complex, and most studies focus on simulations

Balancing a Tree

- Two arguments have been presented in favor of trees:
  - They represent hierarchical data particularly well
  - Searching trees is much faster than searching lists
- However, this second point depends on the structure of the tree
- As we've seen, skewed trees search no better than linear lists
- Three situations are presented in Figure 6.34
  - A fairly well-balanced tree (Figure 6.34a)
  - A right unbalanced tree (Figure 6.34b)
  - A right skewed tree (Figure 6.34c)
- Neither of the last two situations occur in ***balanced trees***

Balancing a Tree (continued)

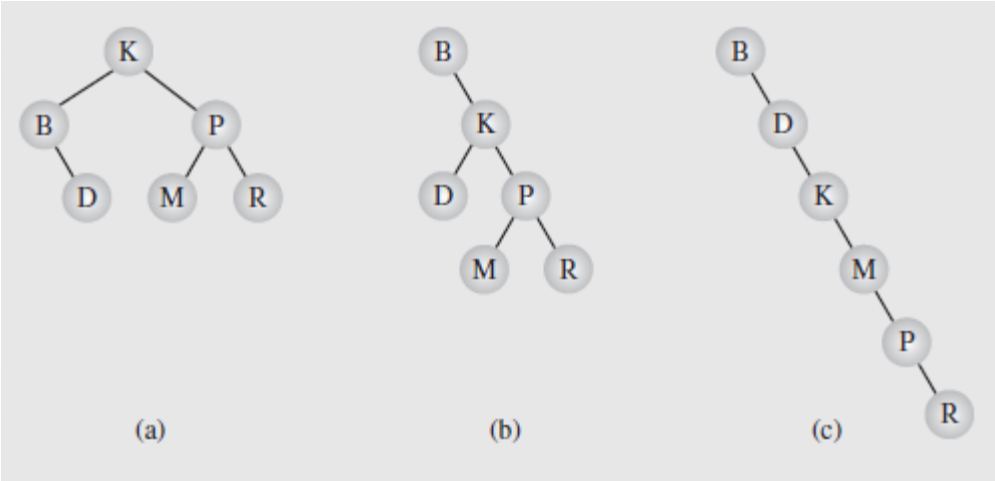
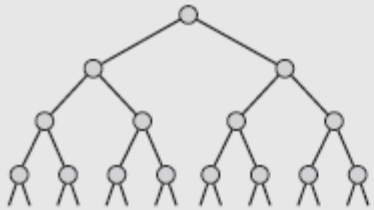


Fig. 6.34 Different binary search trees with the same information

- A binary tree is ***height balanced*** (or simply, ***balanced***) if the difference in height of the subtrees of any node in the tree is zero or one
- It is ***perfectly balanced*** if it is balanced and all leaves are on one or two levels

Balancing a Tree (continued)

- The number of nodes that can be stored in binary trees of different heights is shown in Figure 6.35



Height	Nodes at One Level	Nodes at All Levels
1	$2^0 = 1$	$1 = 2^1 - 1$
2	$2^1 = 2$	$3 = 2^2 - 1$
3	$2^2 = 4$	$7 = 2^3 - 1$
4	$2^3 = 8$	$15 = 2^4 - 1$
⋮		
11	$2^{10} = 1{,}024$	$2{,}047 = 2^{11} - 1$
⋮		
14	$2^{13} = 8{,}192$	$16{,}383 = 2^{14} - 1$
⋮		
h	2^{h-1}	$n = 2^h - 1$
⋮		

Fig. 6.35 Maximum number of nodes in binary trees of different heights

- From this we can see that if we store n elements in a perfectly balanced tree, the height is $\lceil \lg(n+1) \rceil$

Balancing a Tree (continued)

- So storing 10,000 items in such a tree gives us a height of $\lceil \lg(10{,}001) \rceil = \lceil 13.288 \rceil = 14$
- From the standpoint of searching, this means if 10000 items are stored in a perfectly balanced tree, we'll need to look at 14 items to find a particular one
- To find the item in an equivalent linked list would require 10,000 tests in the worst case
- So constructing a balanced tree, or modifying one to make it balanced, is worth the effort

Balancing a Tree (continued)

- AVL Trees
  - The balancing algorithms we've looked at so far can potentially involve the entire tree
  - However, rebalancing can be performed locally if the insertions or deletions impact only a portion of the tree
  - One well-known method is based on the work of Adel'son-Vel'skii and Landis, and is named for them: the AVL tree
  - An ***AVL tree*** (also called an ***admissible tree***) is one where the height of the left and right subtrees of every node differ by at most one
  - Examples of AVL trees are shown in Figure 6.40
  - Numbers in the nodes are the ***balance factors***, which is the difference between the height of the right and left subtrees and should be +1, 0, or -1 for AVL trees

Balancing a Tree (continued)

- AVL Trees (continued)

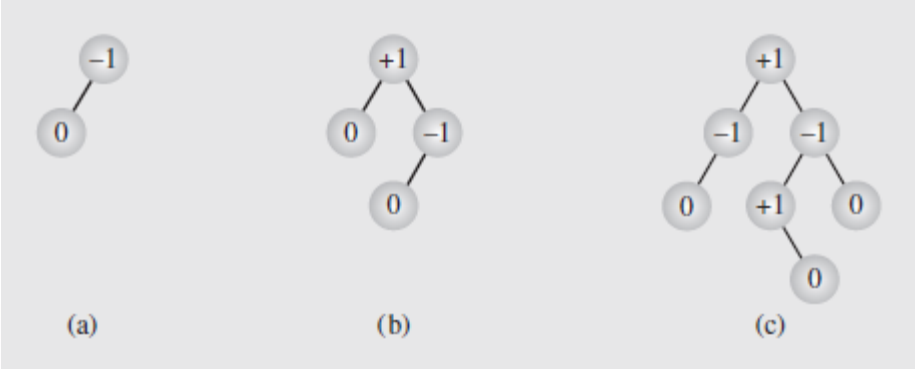


Fig. 6.40 Examples of AVL trees

  - Notice that while the definition of an AVL tree is the same as that of a balanced tree, the model implicitly includes the balancing techniques
  - In addition, the process of balancing an AVL tree does not guarantee a perfectly balanced tree
  - The minimum number of nodes in an AVL tree is determined by:

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

Balancing a Tree (continued)

- AVL Trees (continued)
  - In this recurrence relation, the initial values are $AVL_0 = 0$ and $AVL_1 = 1$
  - From this, we can derive the bounds on the height (h) of the AVL tree based on the number of nodes (n):

$$\lg(n + 1) \leq h < 1.44 \lg(n + 2) - 0.328$$

  - Recall that for a perfectly balanced tree, $h = \lceil \lg(n + 1) \rceil$
  - If any node in an AVL tree has its balance factor become less than -1 or greater than 1, it has to be balanced
  - There are four situations in which a tree can become unbalanced; two are symmetrical to the other two, so we only need to look at two cases
  - These two cases are illustrated in Figure 6.41 and 6.42

Balancing a Tree (continued)

- AVL Trees (continued)
  - The first case, shown in Figure 6.41, occurs when a node is inserted in the right subtree of the right child

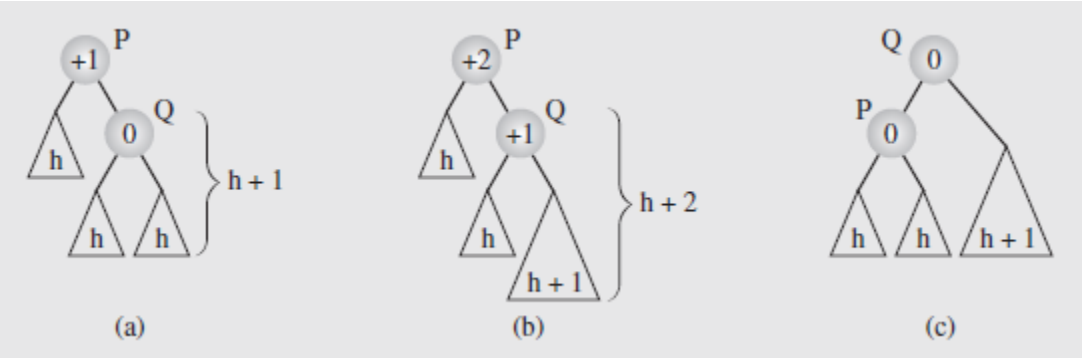


Fig. 6.41 Balancing a tree after insertion of a node in the right subtree of node Q

  - The subtrees involved in the rotation have their heights indicated
  - After a new node is inserted somewhere in the right subtree of *Q* to unbalance the tree, *Q* rotates around is parent *P* to rebalance the tree
  - This is illustrated in Figure 6.41b and Figure 6.41c

Balancing a Tree (continued)

- AVL Trees (continued)
  - The second case, shown in Figure 6.42, occurs when a node is inserted in the left subtree of the right child, and is more complicated

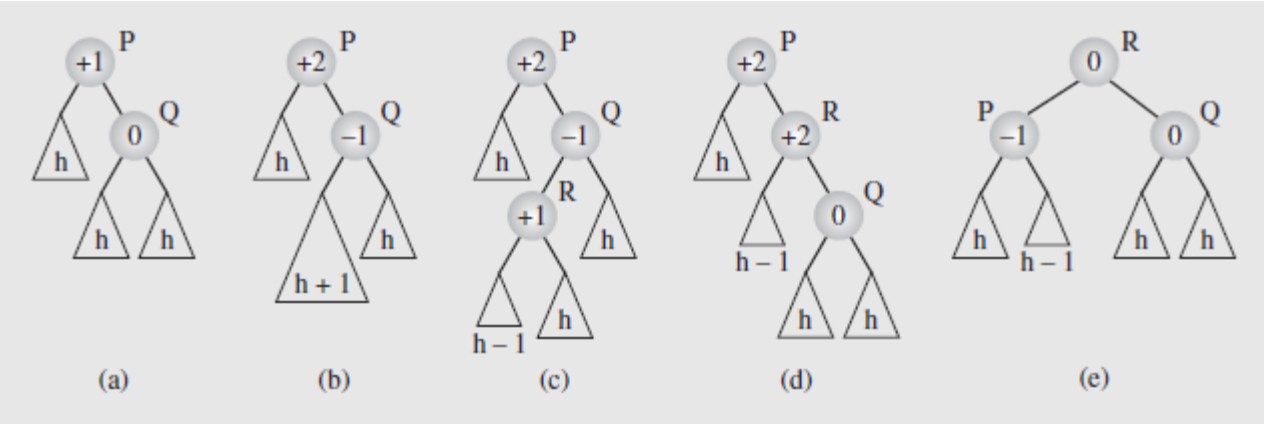


Fig. 6.42 Balancing a tree after insertion of a node in the left subtree of node Q

  - A node is inserted in Figure 6.42a, resulting in the tree in Figure 6.42b
  - The derail of this is in Figure 6.42c; note the subtrees of *R* could be reversed, giving *R* a value of -1
  - The imbalance is solved by a double rotation: first *R* around *Q* (Figure 6.42d), then *R* around *P* (Figure 6.42e)

Balancing a Tree (continued)

- AVL Trees (continued)
  - Figure 6.43 illustrates what happens when a node is inserted and the resulting updates cause a node to become imbalanced

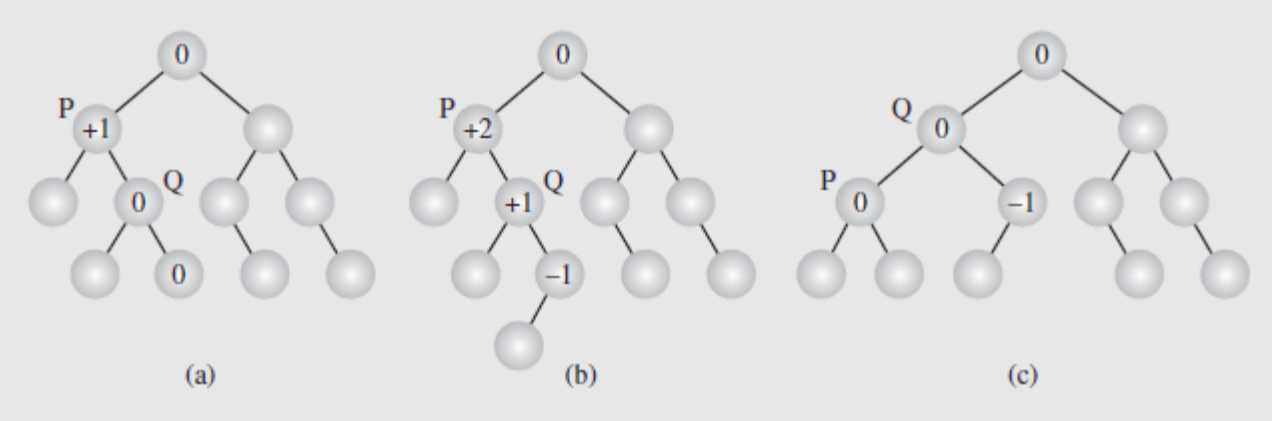


Figure 6.43 An example of inserting (b) a new node in (a) an AVL tree, which requires one rotation (c) to restore the height balance

  - It is also possible that an insertion will only cause the balance factors to be updated; in this case the path is followed back to the root
  - This is shown in Figure 6.44

Balancing a Tree (continued)

- AVL Trees (continued)

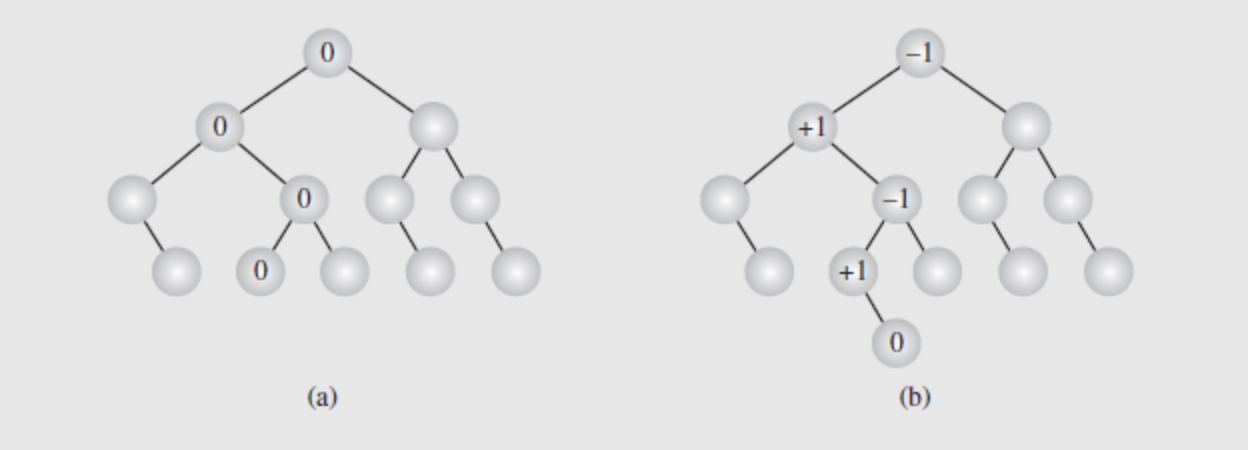


Fig. 6.44 In an (a) AVL tree a (b) new node is inserted requiring no height adjustments

  - Deleting a node may require more work
  - The `deleteByCopying()` algorithm is applied to delete the node
  - The balance factors of all nodes from the parent of the deleted node to the root are then updated
  - Each node whose balance factor becomes ± 2 will be rebalanced
  - This must be done for the entire path because deletion may not cause an imbalance in a parent, but in a grandparent